Deriving Escape Analysis by Abstract Interpretation: Proofs of results

Patricia M. Hill (hill@comp.leeds.ac.uk)
University of Leeds, United Kingdom

Fausto Spoto (fausto.spoto@univr.it)
Università di Verona, Italy

Abstract. Escape analysis of object-oriented languages approximates the set of objects which do not *escape* from a given context. If we take a method as context, the non-escaping objects can be allocated on its activation stack; if we take a thread, Java synchronisation locks on such objects are not needed. In this paper, we formalise a basic escape domain $\mathcal{E}$ as an abstract interpretation of concrete states, which we then refine into an abstract domain $\mathcal{ER}$ which is more concrete than $\mathcal{E}$ and, hence, leads to a more precise escape analysis than $\mathcal{E}$. We provide optimality results for both $\mathcal{E}$ and $\mathcal{ER}$, in the form of Galois insertions from the concrete to the abstract domains and of optimal abstract operations. The Galois insertion property is obtained by restricting the abstract domains to those elements which do not contain *garbage*, by using an abstract garbage collector. Our implementation of $\mathcal{ER}$ is hence an implementation of a formally correct escape analyser, able to detect the stack allocatable creation points of Java (bytecode) applications.

This report contains the proofs of results of a paper with the same title and authors and to be published in the Journal *Higher-Order Symbolic Computation.*

Keywords: Abstract Interpretation, Denotational Semantics, Garbage Collection

1. Introduction

Escape analysis identifies, at compile-time, some run-time data structures which do not *escape* from a given context, in the sense that they are not reachable anymore from that context. It has been studied for functional [24, 13, 4] as well as for object-oriented languages [27, 1, 6, 36, 14, 26, 32, 25, 28, 34, 5, 8, 35]. It allows one to stack allocate dynamically created data structures which would normally be heap allocated. This is possible if these data structures do not *escape* from the method which created them. Stack allocation reduces garbage collection overhead at run-time *w.r.t.* heap allocation, since stack allocated data structures are automatically deallocated when methods terminate. If, moreover, such data structures do not occur in a loop and their size is statically determined, they can be preallocated on the activation stack, which further improves the efficiency of the code. In the case of Java, which uses a mutual exclusion lock for each object in order to synchronise accesses from different threads of execution, escape analysis allows one also to remove unnecessary synchronisations, thereby making run-time accesses faster. By removing the space for the mutual exclusion lock associated with some of the objects, escape analysis can also help with space constraints. To this purpose, the analysis must prove that an object is accessed by at most one thread. This is possible if the object does not *escape* its creating thread.

1.1. Contributions of Our Work

This paper presents two escape analyses for Java programs. The goal of both analyses is to detect objects that do not escape (*i.e.,* are unreachable from outside) a certain scope. This information can later be used to stack-allocate captured (*i.e.,* non-escaping) objects.

Both analyses use the object allocation site model: all objects allocated at a given program point (possibly in a loop) are modelled by the same creation point. The first analysis, based on the abstract domain $\mathcal{E}$, expresses the information we need for our stack allocation. Namely, for each program point, it provides an over-approximation of the set of creation points that escape because they are transitively reachable from a set of escapability roots (*i.e.,* variables including parameters, static fields, method result). The domain $\mathcal{E}$ does not keep track of other information such as the creation points pointed to by each individual variable or field.

Although $\mathcal{E}$ is the property neede for stack allocation, a static analysis based on $\mathcal{E}$ is not sufficiently precise as it does not relate the creation points with the variables and fields that point to them. We therefore consider a refinement $\mathcal{ER}$ of $\mathcal{E}$ that preserves this information and also includes $\mathcal{E}$ so that $\mathcal{ER}$ contains just the minimum information needed for stack allocation.

Both analyses are developed in the abstract interpretation framework [10, 11], and we present proofs that the associated transfer functions are optimal with respect to the abstractions that are used by each analysis *i.e.,* they make the best possible use of the abstract information expressed by the abstract domains.

To increase the precision of the two analyses and to get a Galois insertion, rather than a Galois connection, both analyses use local variable scoping and type information. Hence, the abstract domains contain no spurious element. We achieve this goal through *abstract garbage collectors* which remove some elements from the abstract domains whenever they reflect unreachable (and hence, for our analysis, irrelevant) portions of the run-time heap, as also [8] does, although [8] does not relate this to the Galois insertion property. Namely, the abstract domains are exactly the set of fixpoints of their respective abstract garbage collectors and, hence, do not contain spurious elements.

The contribution of this paper is a clean construction of an escape analysis through abstract interpretation thus obtaining formal and detailed proofs of correctness as well as optimality. Optimality states that the abstract domains are related to the concrete domain by a Galois *insertion*, rather than just a *connection* and in the use of optimal abstract operations. Precision and efficiency of the analysis are not the main issues here, although we are pleased to see that our implementation scales to relatively large applications and compares well with some already existing and more precise escape analyses (Section 6).

1.2. The Basic Domain $\mathcal{E}$

Our work starts by defining a basic abstract domain $\mathcal{E}$ for escape analysis. Its definition is guided by the observation that a creation point π occurring in a method m can be stack allocated if the objects it creates are not reachable at the end of m from a set of variables E which includes m's return value, the fields of the objects bound to its formal parameters at call-time (including the implicit `this` parameter) and any exceptions thrown by m. Note that we consider the fields of the objects bound to the formal parameters at call-time since they are aliases of the actual arguments, and hence still reachable when the method returns. For a language, such as Java, which allows static fields, E also includes the static fields. Variables with integer type are not included in E since no object can be reached from an integer. Moreover, local variables are also not included in E since local variables accessible inside a method m will disappear once m terminates. The basic abstract domain $\mathcal{E}$ is hence defined as the collection of all sets of creation points. Each method is decorated with an element of $\mathcal{E}$, which contains precisely the creation points of the objects reachable from the variables in E at the end of the method.

Example 1 *See journal version of this paper.*

We still have to specify how this decoration is computed for each method. We use abstract interpretation to propagate an input set of creation points through the statements of each method, until its end is reached. This is accomplished by defining a *transfer function* for every statement of the program which, in terms of abstract interpretation, is called an *abstract operation* (see Section 4 and Figure 9). The element of $\mathcal{E}$ resulting at the end of each method is then *restricted* to the appropriate set E for that method through an abstract operation called restrict. By applying the theory of abstract interpretation, we know that this restriction is a conservative approximation of the actual decoration we need at the end of each method.

EXAMPLE 2 *See journal version of this paper.*

The problem here is that although the abstract domain $\mathcal{E}$ expresses the kind of decoration we need for stack allocation, $\mathcal{E}$ has very poor computational properties. In terms of abstract interpretation, it induces very imprecise abstract operations and, just as in the case of the basic domain $\mathcal{G}$ for *groundness analysis* of logic programs [20], it needs refining [15, 29].

We formalise the fact that the approximation in $\mathcal{E}$ can shrink, by means of an *abstract garbage collector* (Definition 25) *i.e.,* a garbage collector that works over sets of creation points instead of concrete objects. When a variable's scope is closed, the abstract garbage collector removes from the approximation of the next statement all creation points which can *only* be reached *from that variable.* The name of abstract garbage collector is justified by the fact that this conservatively maintains in the approximation the creation points of the objects which *might* be reachable in the concrete state, thus modeling in the abstract domain a behaviour similar to that of a concrete garbage collector. It must be noted, however, that our abstract garbage collector only considers reachability from the variables in scope in the current method, while a concrete garbage collector would consider reachability from all variables in the current activation stack.

1.3. THE REFINEMENT $\mathcal{ER}$

The abstract domain $\mathcal{E}$ represents the information we need for stack allocation, but it does not include any other related information that may improve the precision of the abstract operations, such as explicit information about the creation points of the objects bound to *a given* variable or field. However, the ability to reason on a per variable basis is essential for the precision of a static analysis of imperative languages, where assignment to a given variable or field is the basic computational mechanism. So we *refine* $\mathcal{E}$ into a new abstract domain $\mathcal{ER}$ which splits the sets of creation points in $\mathcal{E}$ into subsets, one for each variable or field. We show that $\mathcal{ER}$ strictly contains $\mathcal{E}$, justifying the name of *refinement.*

We perform a static analysis based on $\mathcal{ER}$ exactly as for $\mathcal{E}$ but using the abstract operations for the domain $\mathcal{ER}$ given in Section 5 (see Figure 10).

EXAMPLE 3 *See journal version of this paper.*

The domain $\mathcal{ER}$ can hence be seen as the specification of a new escape analysis, which includes $\mathcal{E}$ as its foundational kernel. Example 3 shows that the abstract domain $\mathcal{ER}$ is actually more precise than $\mathcal{E}$. Our implementation of $\mathcal{ER}$ (Section 6) shows that it can actually be used to obtain non-trivial escape analysis information for Java bytecode.

1.4. Structure of the Paper

After a brief summary of our notation and terminology in Section 2, we pass in Section 3 to recall the framework of [31] on which the analysis is based. Then, in Section 4, we formalise our basic domain $\mathcal{E}$ and provide suitable abstract operations for its analysis. We show that the analysis induced by $\mathcal{E}$ is very imprecise. Hence, in Section 5 we refine the domain $\mathcal{E}$ into the more precise domain $\mathcal{ER}$ for escape analysis. In Section 6, we discuss our prototype implementation and experimental results. Section 7 discusses related work. Section 8 concludes the main part of the paper.

Preliminary, partial versions of this paper appeared in [17] and [18]. The current paper is a seamless fusion of these papers, with the proofs of the theoretical results and with a description and evaluation of the implementation of the escape analysis over the domain $\mathcal{ER}$.

2. Preliminaries

A total (partial) function f is denoted by $\mapsto$ ($\to$). The *domain* (*range*) of f is $\mathsf{dom}(f)$ ($\mathsf{rng}(f)$). We denote by $[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$ the function f where $\mathsf{dom}(f) = \{v_1, \ldots, v_n\}$ and $f(v_i) = t_i$ for $i = 1, \ldots, n$. Its *update* is $f[w_1 \mapsto d_1, \ldots, w_m \mapsto d_m]$, where the domain may be enlarged. By $f|_s$ ($f|_{-s}$) we denote the *restriction* of f to $s \subseteq \mathsf{dom}(f)$ (to $\mathsf{dom}(f) \setminus s$). If f and g are functions, we denote by fg the composition of f and g, such that $fg(x) = f(g(x))$. If $f(x) = x$ then x is a *fixpoint* of f. The set of fixpoints of f is denoted by $\mathsf{fp}(f)$.

A *pair* of elements is written $a \star b$. A definition of a pair S such as $S = a \star b$, with a and b meta-variables, silently defines the pair selectors $s.a$ and $s.b$ for $s \in S$. The cardinality of a set S is denoted by $\#S$. The *disjoint union* of two sets S, T is denoted by $S + T$. To simplify expressions, particulary when the set is used as a subscript, we sometimes write a singleton set $\{x\}$ as x. If S is a set and $\leq$ is a partial relation over S, we say that S is a *partial ordering* if it is reflexive ($s \leq s$ for every $s \in S$), transitive ($s_1 \leq s_2$ and $s_2 \leq s_3$ entail $s_1 \leq s_2$ for every $s_1, s_2, s_3 \in S$) and anti-symmetric ($s_1 \leq s_2$ and $s_2 \leq s_1$ entail $s_1 = s_2$ for every $s_1, s_2 \in S$). If S is a set and $\leq$ a partial ordering on S, then the pair $S \star \leq$ is a *poset*.

A *complete lattice* is a poset $C \star \leq$ where *least upper bounds* (lub) and *greatest lower bounds* (glb) always exist. Let $C \star \leq$ and $A \star \preceq$ be posets and $f : C \mapsto A$. We say that f is *monotonic* if $c_1 \leq c_2$ entails $f(c_1) \preceq f(c_2)$. It is *(co-)additive* if it preserves lub's (glb's). Let $f : A \mapsto A$. The map f is *reductive* (respectively, *extensive*) if $f(a) \preceq a$ (respectively, $a \preceq f(a)$) for any $a \in A$. It is *idempotent* if $f(f(a)) = f(a)$ for any $a \in A$. It is a *lower closure operator* (*lco*) if it is *monotonic*, *reductive* and *idempotent*.

We recall now the basics of abstract interpretation [10, 11]. Let $C \star \leq$ and $A \star \preceq$ be two posets (the concrete and the abstract domain). A *Galois connection* is a pair of monotonic maps $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ such that $\gamma\alpha$ is extensive and $\alpha\gamma$ is reductive. It is a *Galois insertion* when $\alpha\gamma$ is the identity map *i.e.,* when the abstract domain does not contain *useless* elements. If C and A are complete lattices and α is strict and additive, then α is the abstraction map of a Galois connection. If, moreover, α is onto or γ is one-to-one, then α is the abstraction map of a Galois insertion. In a Galois connection, γ can be defined in terms of α as $\gamma(a) = \cup\{c \mid \alpha(c) \preceq a\}$, where $\cup$ is the least upper bound operation over the concrete domain C. Hence, it is enough to provide α to define a Galois connection. An abstract operator $\hat{f} : A^n \mapsto A$ is *correct w.r.t.* $f : C^n \rightarrow C$ if $\alpha f \gamma \preceq \hat{f}$. For each operator f, there exists an *optimal* (most precise) correct abstract operator $\hat{f}$ defined as $\hat{f} = \alpha f \gamma$. This means that $\hat{f}$ does the best it can with the information expressed by the abstract domain. The composition of correct operators is correct. The composition of optimal operators is not necessarily optimal. The *semantics* of a program is the fixpoint of a map $f : C \mapsto C$, where C is the *computational domain.* Its *collecting version* [10, 11] works over *properties* of C *i.e.,* over $\wp(C)$ and is the fixpoint of the powerset extension of f. If f is defined through suboperations, their powerset extensions *and* $\cup$ (which merges the semantics of the branches of a conditional) induce the extension of f.

3. The Framework of Analysis

The framework presented here is for a simple typed object-oriented language where the concrete states and operations are based on [31]. It allows us to derive a compositional, denotational semantics, which can be seen as an analyser, from a specification of a domain of abstract states and operations which work over them (hence called *state transformers*). Then problems such as scoping, recursion and name clash can be ignored, since these are already solved by the semantics. Moreover, this framework relates the precision of the analysis to that of its abstract domain so that traditional techniques for comparing the precision of abstract domains can be applied [9, 10, 11].

The definition of a denotational semantics, in the style of [37], by using the state transformers of this section can be found in [31]. Here we only want to make clear some points:

– We allow expressions to have side-effects, such as method call expressions, which is not the case in [37]. As a consequence, the evaluation of an expression from an initial state yields both a final state *and* the value of the expression. We use a special variable *res* of the final state to hold this value;

– The evaluation from an initial state σ_1 of a binary operation such as $e_1 + e_2$, where e_1 and e_2 are expressions, first evaluates e_1 from σ_1, yielding an intermediate state σ_2, and then evaluates e_2 from σ_2, yielding a state σ_3. The value v_1 of *res* in σ_2 is that of e_1, and the value v_2 of *res* in σ_3 is that of e_2. We then modify σ_3 by storing in *res* the sum $v_1 + v_2$. This yields the final state. Note that the single variable *res* is enough for this purpose. The complexity of this mechanism *w.r.t.* a more standard approach [37] is, again, a consequence of the use of expressions with side-effects;

– Our denotational semantics deals with method calls through *interpretations*: an interpretation is the input/output behaviour of a method, and is used as its denotation whenever that method is called. As a nice consequence, our states contain only a single frame, rather than an activation stack of frames. This is standard in denotational semantics and has been used for years in logic programming [7].

– The computation of the semantics of a program starts from a bottom interpretation which maps every input state to an undefined final state and then updates this interpretation with the denotations of the methods body. This process is iterated until a fixpoint is reached as is done for logic programs [7]. The same technique can be applied to compute the abstract semantics of a program, but the computation is performed over the abstract domain. It is also possible to generate constraints which relate the abstract approximations at different program points, and then solve such constraints with a fixpoint engine. The latter is the technique that we use in Section 6.

3.1. Programs and Creation Points

We recall here the semantical framework of [31].

Definition 4 (Type Environment) *Each program in the language has a finite set of* identifiers *Id such that* $\mathtt{out}, \mathtt{this} \in \mathit{Id}$ *and a finite set of* classes $\mathcal{K}$ *ordered by a* subclass relation $\leq$ *such that* $\mathcal{K} \star \leq$ *is a poset. Let* $\mathit{Type} = \{\mathit{int}\} \uplus \mathcal{K}$ *and* $\leq$ *be extended to Type by defining* $\mathit{int} \leq \mathit{int}$. *Let* $\mathit{Vars} \subseteq \mathit{Id}$ *be a set of* variables *such that* $\{\mathtt{out}, \mathtt{this}\} \subseteq \mathit{Vars}$. *A* type environment *for a program is any element of the set*

$$\mathit{TypEnv} = \left\{\tau : \mathit{Vars} \to \mathit{Type} \;\middle|\; \mathit{if}\ \mathtt{this} \in \mathsf{dom}(\tau)\ \mathit{then}\ \tau(\mathtt{this}) \in \mathcal{K}\ \right\}.$$

In the following, τ *will implicitly stand for a type environment.*

A class contains local variables (*fields*) and functions (*methods*). A method has a set of input/output variables called *parameters*, including `out`, which holds the result of the method, and `this`, which is the object over which the method has been called (the *receiver* of the call). Methods returning `void` are represented as methods returning an *int* of constant value 0, implicitly ignored by the caller of the method.

Example 5 *See journal version of this paper.*

Fields is a set of maps which bind each class to the type environment of its fields. The variable `this` cannot be a field. *Methods* is a set of maps which bind each class to a map from identifiers to methods. *Pars* is a set of maps which bind each method to the type environment of its parameters (its signature).

Definition 6 (Field, Method, Parameter) *Let* $\mathcal{M}$ *be a finite set of* methods. *We define*

$$\begin{aligned} \mathit{Fields} &= \{F : \mathcal{K} \mapsto \mathit{TypEnv} \mid \mathtt{this} \notin \mathsf{dom}(F(\kappa))\ \mathit{for\ every}\ \kappa \in \mathcal{K}\} \\ \mathit{Methods} &= \mathcal{K} \mapsto (\mathit{Id} \to \mathcal{M}) \\ \mathit{Pars} &= \{P : \mathcal{M} \mapsto \mathit{TypEnv} \mid \{\mathtt{out}, \mathtt{this}\} \subseteq \mathsf{dom}(P(\nu))\ \mathit{for}\ \nu \in \mathcal{M}\}. \end{aligned}$$

The *static information* of a program is used by the static analyser.

Definition 7 (Static Information) *The* static information *of a program consists of a poset* $\mathcal{K} \star \leq$, *a set of methods* $\mathcal{M}$ *and maps* $F \in \mathit{Fields}$, $M \in \mathit{Methods}$ *and* $P \in \mathit{Pars}$.

Fields in different classes but with the same name can be disambiguated by using their *fully qualified name* such as in the Java Virtual Machine [21]. For instance, we write `Circle.x` for the field `x` of the class `Circle`.

Example 8 *See journal version of this paper.*

The only points in the program where new objects can be created are the `new` statements. We require that each of these statements is identified by a unique label called its *creation point*.

DEFINITION 9 (CREATION POINT) *Let* Π *be a finite set of labels called* creation points*. A map* $k : \Pi \mapsto \mathcal{K}$ *relates every creation point* $\pi \in \Pi$ *with the class* $k(\pi)$ *of the objects it creates.*

EXAMPLE 10 *See journal version of this paper.*

3.2. CONCRETE STATES

To represent the concrete state of a computation at a particular program point we need to refer to the concrete values that may be assigned to the variables. Apart from the integers and *null*, these values need to include *locations* which are the addresses of the memory cells used at that point. Then the concrete state of the computation consists of a map that assigns type consistent values to variables (*frame*) and a map from locations to objects (*memory*) where an *object* is characterised by its creation point and the frame of its fields. Hence the notion of object that we use here is more concrete than that in [31], which relates a *class* rather than a *creation point* to each object. A memory can be *updated* by assigning new (type consistent) values to the variables in its frames.

DEFINITION 11 (LOCATION, FRAME, OBJECT, MEMORY) *Let Loc be an infinite set of* locations *and* $Value = \mathbb{Z} + Loc + \{null\}$*. We define* frames, objects *and* memories *as*

$$
\begin{aligned}
Frame_\tau &= \left\{ \phi \in \mathsf{dom}(\tau) \mapsto Value \;\middle|\; \begin{array}{l} \text{for every } v \in \mathsf{dom}(\tau) \\ \tau(v) = int \Rightarrow \phi(v) \in \mathbb{Z} \\ \tau(v) \in \mathcal{K} \Rightarrow \phi(v) \in \{null\} \cup Loc \end{array} \right\} \\
Obj &= \{\pi \star \phi \mid \pi \in \Pi,\ \phi \in Frame_{F(k(\pi))}\} \\
Memory &= \{\mu \in Loc \to Obj \mid \mathsf{dom}(\mu) \text{ is finite}\}.
\end{aligned}
$$

Let $\mu_1, \mu_2 \in Memory$ *and* $L \subseteq \mathsf{dom}(\mu_1)$*. We say that* μ_2 *is an* L-update *of* μ_1*, written* $\mu_1 =_L \mu_2$*, if* $L \subseteq \mathsf{dom}(\mu_2)$ *and for every* $l \in L$ *we have* $\mu_1(l).\pi = \mu_2(l).\pi$.

The initial value for a variable of a given type is used when we add a variable in scope. It is defined as $\Im(int) = 0$*,* $\Im(\kappa) = null$ *for* $\kappa \in \mathcal{K}$*. This function is extended to type environments (Definition 4) as* $\Im(\tau)(v) = \Im(\tau(v))$ *for every* $v \in \mathsf{dom}(\tau)$*.*

EXAMPLE 12 *See journal version of this paper.*

Type correctness and conservative garbage collection guarantee that there are no dangling pointers and that variables may only be bound to locations which contain objects allowed by the type environment. This is a sensible constraint for the memory allocated by strongly-typed languages such as Java [2].

DEFINITION 13 (WEAK CORRECTNESS) *Let $\phi \in Frame_\tau$ and $\mu \in Memory$. We say that ϕ is* weakly τ-correct *w.r.t. μ if for every $v \in \mathsf{dom}(\phi)$ such that $\phi(v) \in Loc$ we have $\phi(v) \in \mathsf{dom}(\mu)$ and $k((\mu\phi(v)).\pi) \leq \tau(v)$.*

We strengthen the correctness notion of Definition 13 by requiring that it also holds for the fields of the objects in memory.

DEFINITION 14 (τ-CORRECTNESS) *Let $\phi \in Frame_\tau$ and $\mu \in Memory$. We say that ϕ is* τ-correct *w.r.t. μ and write $\phi \star \mu : \tau$, if*

1. *ϕ is weakly τ-correct w.r.t. μ and,*
2. *for every $o \in \mathsf{rng}(\mu)$, $o.\phi$ is weakly $F(k(o.\pi))$-correct w.r.t. μ.*

EXAMPLE 15 *See journal version of this paper.*

Definition 16 defines the state of the computation as a pair consisting of a frame and a memory. The variable `this` in the domain of the frame must be bound to an object. In particular, it cannot be *null*. This condition could be relaxed in Definition 16. This would lead to simplifications in the following sections (such as in Definition 25). However, our condition is consistent with the specification of the Java programming language [2]. Note, however, that there is no such hypothesis about the local variable number 0 of the Java Virtual Machine, which stores the `this` object [21].

DEFINITION 16 (STATE) *If τ is a type environment associated with a program point, the set of possible* states *of a computation at that point is any subset of*

$$\Sigma_\tau = \left\{ \phi \star \mu \;\middle|\; \begin{array}{l} \phi \in Frame_\tau,\ \mu \in Memory,\ \phi \star \mu : \tau, \\ \textit{if } \mathtt{this} \in \mathsf{dom}(\tau) \textit{ then } \phi(\mathtt{this}) \neq null \end{array} \right\}.$$

EXAMPLE 17 *See journal version of this paper.*

The frame of an object o in memory is itself a state for the instance variables of o.

PROPOSITION 18 *Let* $\phi \star \mu \in \Sigma_\tau$ *and* $o \in \mathsf{rng}(\mu)$*. Then* $(o.\phi) \star \mu \in \Sigma_{F(k(o.\pi))}$.

Proof. Since $\phi \star \mu \in \Sigma_\tau$, from Definition 16 we have $\phi \star \mu : \tau$. From Definition 14 we know that $o.\phi$ is weakly $F(k(o.\pi))$-correct *w.r.t.* μ so that $(o.\phi) \star \mu : F(k(o.\pi))$. Since `this` $\notin \mathsf{dom}(F(k(o.\pi)))$ (Definition 6) we conclude that $(o.\phi) \star \mu \in \Sigma_{F(k(o.\pi))}$. □

3.3. The Operations over the Concrete States

Figures 7 and 8 show the signatures and the definitions, respectively, of a set of operations over the concrete states for a type environment τ. The variable *res* holds intermediate results, as we said at the beginning of this section. We briefly introduce these operations.

- The nop operation does nothing.
- A get operation loads into *res* a constant, the value of another variable or the value of the field of an object. In the last case (get_field), that object is assumed to be stored in *res* *before* the get operation. Then $(\mu\phi'(res))$ is the object whose field f must be read, $(\mu\phi'(res)).\phi$ are its fields and $(\mu\phi'(res)).\phi(f)$ is the value of the field named f.
- A put operation stores in v the value of *res* or of a field of an object pointed to by *res*. Note that, in the second case, put_field is a binary operation since the evaluation of $e_1.f = e_2$ from an initial state σ_1 works by first evaluating e_1 from σ_1, yielding an intermediate state σ_2, and then evaluating e_2 from σ_2, yielding a state σ_3. The final state is then $\mathsf{put\_field}(\sigma_2)(\sigma_3)$ [31], where the variable *res* of σ_2 holds the value of e_1 and the variable *res* of σ_3 holds the value of e_2. The object whose field is modified must still exist in the memory of σ_3. This is expressed by the update relation (Definition 11). As there is no result, *res* is removed. Providing two states *i.e.,* two frames and two heaps for put_field and, more generally, for binary operations, may look like an overkill and it might be expected that a single state and a single frame would be enough. However, our decision to have two states has been dictated by the intended use of this semantics *i.e.,* abstract interpretation. By *only* using operations over states, we have exactly one concrete domain, which can be abstracted into just one abstract domain. Hybrid operations, working on states and frames, would only complicate the abstraction.

Operation	Constraint ($\texttt{this} \in \mathsf{dom}(\tau)$ always)
$\mathsf{nop}_\tau : \Sigma_\tau \mapsto \Sigma_\tau$	
$\mathsf{get\_int}^i_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[res \mapsto int]}$	$res \notin \mathsf{dom}(\tau),\ i \in \mathbb{Z}$
$\mathsf{get\_null}^\kappa_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[res \mapsto \kappa]}$	$res \notin \mathsf{dom}(\tau),\ \kappa \in \mathcal{K}$
$\mathsf{get\_var}^v_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[res \mapsto \tau(v)]}$	$res \notin \mathsf{dom}(\tau),\ v \in \mathsf{dom}(\tau)$
$\mathsf{get\_field}^f_\tau : \Sigma_\tau \to \Sigma_{\tau[res \mapsto i(f)]}$	$res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathcal{K},$ $i = F\tau(res),\ f \in \mathsf{dom}(i)$
$\mathsf{put\_var}^v_\tau : \Sigma_\tau \mapsto \Sigma_{\tau\|_{-res}}$	$res \in \mathsf{dom}(\tau),\ v \in \mathsf{dom}(\tau),$ $v \neq res,\ \tau(res) \leq \tau(v)$
$\mathsf{put\_field}^f_{\tau,\tau'} : \Sigma_\tau \mapsto \Sigma_{\tau'} \to \Sigma_{\tau\|_{-res}}$	$res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathcal{K}$ $f \in \mathsf{dom}(F\tau(res))$ $\tau' = \tau[res \mapsto t]$ with $t \leq (F\tau(res))(f)$
$=_\tau, +_\tau : \Sigma_\tau \mapsto \Sigma_\tau \mapsto \Sigma_\tau$	$res \in \mathsf{dom}(\tau),\ \tau(res) = int$
$\mathsf{is\_null}_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[res \mapsto int]}$	$res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathcal{K}$
$\mathsf{call}^{\nu,v_1,\ldots,v_n}_\tau : \Sigma_\tau \mapsto \Sigma_{P(\nu)\|_{-\texttt{out}}}$	$res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathcal{K},$ $\{v_1,\ldots,v_n\} \subseteq \mathsf{dom}(\tau),\ \nu \in \mathcal{M}$ $\mathsf{dom}(P(\nu)) \setminus \{\texttt{out},\texttt{this}\} = \{\iota_1,\ldots,\iota_n\}$ (alphabetically ordered) $\tau(res) \leq P(\nu)(\texttt{this})$ $\tau(v_i) \leq P(\nu)(\iota_i)$ for $i = 1,\ldots,n$
$\mathsf{return}^\nu_\tau : \Sigma_\tau \mapsto \Sigma_{p\|_{\texttt{out}}} \to \Sigma_{\tau[res \mapsto p(\texttt{out})]}$	$res \in \mathsf{dom}(\tau),\ \nu \in \mathcal{M},\ p = P(\nu)$
$\mathsf{restrict}^{vs}_\tau : \Sigma_\tau \mapsto \Sigma_{\tau\|_{-vs}}$	$vs \subseteq \mathsf{dom}(\tau)$
$\mathsf{expand}^{v:t}_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[v \mapsto t]}$	$v \in Vars,\ v \notin \mathsf{dom}(\tau),\ t \in Type$
$\mathsf{new}^\pi_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[res \mapsto k(\pi)]}$	$res \notin \mathsf{dom}(\tau),\ \pi \in \Pi$
$\mathsf{lookup}^{m,\nu}_\tau : \Sigma_\tau \to \Sigma_{\tau[res \mapsto P(\nu)(\texttt{this})]}$	$res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathcal{K},$ $m \in \mathsf{dom}(M\tau(res)),\ \nu \in \mathcal{M}$ for every suitable m, σ and τ, there is at most one ν such that $\mathsf{lookup}^{m,\nu}_\tau(\sigma)$ is defined
$\mathsf{is\_true}_\tau : \Sigma_\tau \to \Sigma_{\tau\|_{-res}}$ $\mathsf{is\_false}_\tau : \Sigma_\tau \to \Sigma_{\tau\|_{-res}}$	$res \in \mathsf{dom}(\tau),\ \tau(res) = int,$ $\mathsf{dom}(\mathsf{is\_true}_\tau) \cap \mathsf{dom}(\mathsf{is\_false}_\tau) = \varnothing$ $\mathsf{dom}(\mathsf{is\_true}_\tau) \cup \mathsf{dom}(\mathsf{is\_false}_\tau) = \Sigma_\tau$

Figure 7. The signature of the operations over the states.

- For every binary operation such as $=$ and $+$ over values, there is an operation on states. Note that (in the case of $=$) Booleans are implemented by means of integers (every non-negative integer means true). We have already explained why we use two states for binary operations.

- The operation is_null checks that *res* points to *null*.

- The operation call is used before, and the operation return is used after, a call to a method ν. While call^ν creates a new state in

$$
\begin{aligned}
\mathsf{nop}_\tau(\phi \star \mu) &= \phi \star \mu\\
\mathsf{get\_int}^i_\tau(\phi \star \mu) &= \phi[res \mapsto i] \star \mu\\
\mathsf{get\_null}^\kappa_\tau(\phi \star \mu) &= \phi[res \mapsto null] \star \mu\\
\mathsf{get\_var}^v_\tau(\phi \star \mu) &= \phi[res \mapsto \phi(v)] \star \mu\\
\mathsf{restrict}^{vs}_\tau(\phi \star \mu) &= \phi|_{-vs} \star \mu\\
\mathsf{expand}^{v:t}_\tau(\phi \star \mu) &= \phi[v \mapsto \Im(t)] \star \mu\\
\mathsf{put\_var}^v_\tau(\phi \star \mu) &= \phi[v \mapsto \phi(res)]|_{-res} \star \mu\\
\mathsf{get\_field}^f_\tau(\phi' \star \mu) &= \begin{cases} \phi'[res \mapsto ((\mu\phi'(res)).\phi)(f)] \star \mu & \text{if } \phi'(res) \neq null\\ \text{undefined} & \text{otherwise}\end{cases}\\
\begin{matrix}\mathsf{put\_field}^f_{\tau,\tau'}\\ (\phi_1 \star \mu_1)(\phi_2 \star \mu_2)\end{matrix} &= \begin{cases} \phi_2|_{-res} \star \mu_2[l \mapsto \mu_2(l).\pi \star \mu_2(l).\phi[f \mapsto \phi_2(res)]]\\ \qquad \text{if } l = \phi_1(res), l \neq null \text{ and } \mu_1 =_l \mu_2\\ \text{undefined} \quad \text{otherwise}\end{cases}\\
=_\tau(\phi_1 \star \mu_1)(\phi_2 \star \mu_2) &= \begin{cases} \phi_2[res \mapsto 1] \star \mu_2 & \text{if } \phi_1(res) = \phi_2(res)\\ \phi_2[res \mapsto -1] \star \mu_2 & \text{if } \phi_1(res) \neq \phi_2(res)\end{cases}\\
+_\tau(\phi_1 \star \mu_1)(\phi_2 \star \mu_2) &= \phi_2[res \mapsto \phi_1(res) + \phi_2(res)] \star \mu_2\\
\mathsf{is\_null}_\tau(\phi \star \mu) &= \begin{cases} \phi[res \mapsto 1] \star \mu & \text{if } \phi(res) = null\\ \phi[res \mapsto -1] \star \mu & \text{otherwise}\end{cases}\\
\mathsf{call}^{\nu, v_1, \ldots, v_n}_\tau(\phi \star \mu) &= [\iota_1 \mapsto \phi(v_1), \ldots, \iota_n \mapsto \phi(v_n), \mathtt{this} \mapsto \phi(res)] \star \mu\\
\text{where } \{\iota_1, \ldots, \iota_n\} &= P(\nu) \setminus \{\mathtt{out}, \mathtt{this}\} \text{ (alphabetically ordered)}\\
\begin{matrix}\mathsf{return}^\nu_\tau\\ (\phi_1 \star \mu_1)(\phi_2 \star \mu_2)\end{matrix} &= \begin{cases} \phi_1[res \mapsto \phi_2(\mathtt{out})] \star \mu_2\\ \quad \text{if } L = \mathsf{rng}(\phi_1)|_{-res} \cap Loc \text{ and } \mu_1 =_L \mu_2\\ \\ \text{undefined} \quad \text{otherwise}\end{cases}\\
\mathsf{new}^\pi_\tau(\phi \star \mu) &= \phi[res \mapsto l] \star \mu[l \mapsto \pi \star \Im(F(k(\pi)))], \; l \in Loc \setminus \mathsf{dom}(\mu)\\
\mathsf{lookup}^{m,\nu}_\tau(\phi \star \mu) &= \begin{cases} \phi \star \mu\\ \quad \text{if } \phi(res) \neq null \text{ and } M(k((\mu\phi(res)).\pi))(m) = \nu\\ \\ \text{undefined} \quad \text{otherwise}\end{cases}\\
\mathsf{is\_true}_\tau(\phi \star \mu) &= \begin{cases} \phi|_{-res} \star \mu & \text{if } \phi(res) \geq 0\\ \text{undefined} & \text{otherwise}\end{cases}\\
\mathsf{is\_false}_\tau(\phi \star \mu) &= \begin{cases} \phi|_{-res} \star \mu & \text{if } \phi(res) < 0\\ \text{undefined} & \text{otherwise.}\end{cases}
\end{aligned}
$$

Figure 8. The operations over concrete states.

which ν can execute, the operation return^ν restores the state σ which was current before the call to ν, and stores in *res* the result of the call. As said in (the beginning of) Section 3, the denotation of the method is taken from an *interpretation*, in a denotational fashion [7]. Hence the execution from an initial state σ_1 of a method call denoted, in the current interpretation, by $d : \Sigma \to \Sigma$, yields the final state $\mathsf{return}(\sigma_1)(d(\mathsf{call}(\sigma_1)))$. Note that return is a binary operation whose first argument is the state of the caller at call-time and whose second argument is the state of the callee at return-time. Its definition in Figure 8 restores the state of the caller but stores in *res* the return value of the callee. By using a binary operation we can define our semantics in terms of states rather than in terms of activation stacks. This is a useful simplification when passing to abstraction, since states must be abstracted rather than stacks. Note that the update relation (Definition 11) requires that the variables of the caller have not been changed during the execution of the method (although the fields of the objects bound to those variables may be changed).

- The operation expand ($\mathsf{restrict}$) adds (removes) variables.
- The operation new^{π} creates a new object o of creation point π. A pointer to o is put in *res*. Its fields are initialised to default values.
- The operation $\mathsf{lookup}^{m,\nu}$ checks if, by calling the method identified by m of the object o pointed to by *res*, the method ν is run. This depends on the class $k(o.\pi)$ of $o = \mu\phi(res)$.
- The operation $\mathsf{is\_true}$ ($\mathsf{is\_false}$) checks if *res* contains true (false).

EXAMPLE 19 *See journal version of this paper.*

3.4. The Collecting Semantics

The operations of Figure 8 can be used to define the transition function from states to states, or *denotation*, of a piece of code c, as shown in Example 19. By use of call and return, there is a denotation for each method called in c; thus, by adding call and return, we can plug the method's denotation in the calling points inside c (as shown in Subsection 3.3 and in Example 19). A function I binding each method $\mathtt{m}$ in a program P to its denotation $I(\mathtt{m})$ is called an *interpretation* of P. Given an interpretation I, we are hence able to define the denotation $T_P(I)(\mathtt{m})$ of the body of a method $\mathtt{m}$, so that we are able to transform I into a new interpretation $T_P(I)$. This leads to the definition of the *denotational semantics* of P as the minimal (*i.e.,* less defined) interpretation which is a fixpoint of T_P. This way of defining the concrete semantics in a denotational way through interpretations, is useful for a subsequent abstraction [11]. The technique, which has been extensively used in the logic programming tradition [7], has been adapted in [31] for object-oriented imperative programs by adding the mechanism for dynamic dispatch through the lookup operation in Figure 8. Note that the fixpoint of T_P is not finitely computable in general, but it does exist as a consequence of Tarski's theorem and it is the limit of the ascending chain of interpretations I_0, $T_P(I_0)$, $T_P(T_P(I_0))$, …, where, for every method `m`, the denotation $I_0(\mathtt{m})$ is always undefined [33].

The concrete semantics described above denotes each method with a map on states *i.e.,* a function from Σ to Σ. However, abstract interpretation is interested in *properties* of states; so that each property of interest, is identified with the set of all the states satisfying that property. This leads to the definition of a *collecting* semantics [10, 11] *i.e.,* a concrete semantics working over the powerset $\wp(\Sigma)$. The operations of this collecting semantics are the powerset extension of the operations in Figure 8. For instance, $\mathsf{get\_int}^i_\tau$ is extended into

$$\mathsf{get\_int}^i_\tau(S) = \{\mathsf{get\_int}^i_\tau(\sigma) \mid \sigma \in S\}$$

for every $S \in \wp(\Sigma_\tau)$. Note that dealing with powersets means that the semantics becomes non-deterministic. For instance, in Example 19 more than one target of the `f.def()` virtual call could be selected at the same time and more than one of the blocks of code could be executed. Hence we need a $\cup$ operation over sets of states which merges different threads of execution at the end of a virtual call (or, for similar motivations, at the end of a conditional). The notion of denotation now becomes a map over $\wp(\Sigma_\tau)$. Interpretations and the transformer on interpretations are defined exactly as above. We will assume the result, proved in [31], that every abstraction of $\wp(\Sigma_\tau)$, $\cup$ and of the powerset extension of the operations in Figure 8 induces an abstraction of the concrete collecting semantics. This is an application to object-oriented imperative programs of the *fixpoint transfer* Proposition 27 in [11]. Two such abstractions will be described in Sections 4 and 5.

4. The Basic Domain $\mathcal{E}$

We define here a basic abstract domain $\mathcal{E}$ as a property of the concrete states of Definition 16. Its definition is guided by our goal to *over*approximate, for every program point p, the set of creation points of objects reachable at p from some variable or field in scope. Thus an element of the abstract domain $\mathcal{E}$ which decorates a program point p is simply a set of creation points of objects that may be reached at p. The choice of an *over*approximation follows from the typical use of the information provided by an escape analysis. For instance, an object can be stack allocated if it does not escape the method which creates it *i.e.,* if it does not belong to a superset of the objects reachable at its end. Moreover, our goal is to stack allocate specific creation points. Hence, we are not interested in the identity of the objects but in their creation points.

Although, at the end of this section, we will see that $\mathcal{E}$ induces rather imprecise abstract operations, its definition is important since $\mathcal{E}$ comprises exactly the information needed to implement our escape analysis. Even though its abstract operations lose precision, we still need $\mathcal{E}$ as a basis for comparison and as a minimum requirement for new, improved domains for escape analysis. Namely, in Section 5 we will define a more precise abstract domain $\mathcal{ER}$ for escape analysis, and we will prove (Proposition 56) that it strictly contains $\mathcal{E}$. This situation is similar to that of the abstract domain $\mathcal{G}$ for groundness analysis of logic programs [30] which, although imprecise, expresses the property looked for by the analysis, and is the basis of all the other abstract domains for groundness analysis, derived as *refinements* of $\mathcal{G}$ [29]. The definition of more precise abstract domains as refinements of simpler ones is actually standard methodology in abstract interpretation nowadays [15]. Another example is strictness analysis of functional programs, where a first simple domain is subsequently enriched to express more precise information [19]. A similar idea has also been applied to model-checking, through a sequence of refinements of a simple abstract domain [12]. A *refinement*, in this context, is just an operation that transforms a simpler domain into a richer one *i.e.,* one containing more abstract elements. There are many standard refinements operations. One of this is *reduced product*, which allows one to compose two abstract domains in order to express the composition of the properties expressed by the two domains, and *disjunctive completion*, which enriches an abstract domain with the ability to express disjunctive information about the properties expressed by the domain [22]. Another example is the *linear refinement* of a domain *w.r.t.* another, which expresses the dependencies of the abstract properties expressed by the two domains [16]. In Section 5 we use a refinement which is significant for imperative programs, where assignments to program variables are the pervasive operation. Hence, a variable-based approximation often yields improved precision *w.r.t.* a global approximation of the state, such as expressed by $\mathcal{E}$. This same refinement is used, for instance, when passing from rapid type analysis to a variable-based *class analysis* of object-oriented imperative programs in [31].

We show an example now that clarifies the idea of *reachability* for objects at a program point.

EXAMPLE 20 *See journal version of this paper.*

The reasoning in Example 20 leads to the notion of *reachability* in Definition 21 where we use the actual fields of the objects instead of those of the declared class of the variables.

Definition 21 (Reachability) *Let $\sigma = \phi \star \mu \in \Sigma_\tau$ and $S \subseteq \Sigma_\tau$. The set of the objects* reachable *in σ is $O_\tau(\sigma) = \cup\{O_\tau^i(\sigma) \mid i \geq 0\}$ where*

$$O_\tau^0(S) = \varnothing$$
$$O_\tau^{i+1}(S) = \bigcup \left\{ \{o\} \cup O^i_{F(k(o.\pi))}(o.\phi \star \mu) \;\middle|\; \begin{array}{l} \phi \star \mu \in S,\ v \in \mathsf{dom}(\tau) \\ \phi(v) \in Loc,\ o = \mu\phi(v) \end{array} \right\}.$$

The maps O_τ^i are extended to $\wp(\Sigma_\tau)$ as $O_\tau^i(S) = \cup\{O_\tau^i(\sigma) \mid \sigma \in S\}$.

Proposition 18 provides a guarantee that Definition 21 is well-defined. Observe that variables and fields of type *int* do not contribute to O_τ. We can now define the abstraction map for $\mathcal{E}$. It selects the creation points of the reachable objects.

Definition 22 (Abstraction Map for $\mathcal{E}$) *Let $S \subseteq \Sigma_\tau$. The* abstraction map *for $\mathcal{E}$ is*

$$\alpha_\tau^{\mathcal{E}}(S) = \{o.\pi \mid \sigma \in S \text{ and } o \in O_\tau(\sigma)\} \subseteq \Pi.$$

Example 23 *See journal version of this paper.*

4.1. The Domain $\mathcal{E}$ in the Presence of Type Information

Definition 22 seems to suggest that $\mathsf{rng}(\alpha_\tau^{\mathcal{E}}) = \wp(\Pi)$ *i.e.,* that every set of creation points is a legal approximation in each given program point. However, this is not true if type information is taken into account.

Example 24 *See journal version of this paper.*

Example 24 shows that *static type information provides escape information* by indicating which subsets of creation points are not the abstraction of any concrete states. We should therefore characterise which are the *good* or meaningful elements of $\wp(\Pi)$. This is important because it reduces the size of the abstract domain and removes useless creation points during the analysis through the use of an *abstract garbage collector* δ_τ (Definition 25).

Let $e \in \wp(\Pi)$. Then $\delta_\tau(e)$ is defined as the largest subset of e which contains only those creation points deemed useful by the type environment τ. This set is computed first by collecting the creation points that create objects compatible with the types in τ. For each of these points, this check is reiterated for each of the fields of the object it creates until a fixpoint is reached. Note that if there are no possible creation points for `this`, all creation points are useless.

Definition 25 (Abstract Garbage Collector δ) *Let $e \subseteq \Pi$. We define $\delta_\tau(e) = \cup\{\delta^i_\tau(e) \mid i \geq 0\}$ with*

$$\delta^0_\tau(e) = \varnothing$$

$$\delta^{i+1}_\tau(e) = \begin{cases} \varnothing \\ \quad \text{if } \mathtt{this} \in \mathsf{dom}(\tau) \text{ and no } \pi \in e \text{ is s.t. } k(\pi) \leq \tau(\mathtt{this}) \\ \\ \cup\left\{ \{\pi\} \cup \delta^i_{F(\pi)}(e) \;\middle|\; \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K},\ \pi \in e,\ k(\pi) \leq \kappa \right\} \\ \quad \text{otherwise.} \end{cases}$$

It follows from Definition 25 that $\delta^i_\tau \subseteq \delta^{i+1}_\tau$ and hence $\delta_\tau = \delta^{\#\Pi}_\tau$. Note that in Definition 25 we consider all subclasses of κ (Example 20).

Example 26 *See journal version of this paper.*

Proposition 27 states that the abstract garbage collector δ_τ is a lower closure operator so that it possesses the properties of monotonicity, reductivity and idempotence that would be expected in a garbage collector.

Proposition 27 *Let $i \in \mathbb{N}$. The abstract garbage collectors δ^i_τ and δ_τ are lco's.*

The following result proves that δ_τ can be used to define $\mathsf{rng}(\alpha^{\mathcal{E}}_\tau)$. Namely, the *useful* elements of $\wp(\Pi)$ are those that do not contain any garbage. The proof of Proposition 28 relies on the explicit construction, for every $e \subseteq \Pi$, of a set of concrete states X such that $\alpha_\tau(X) = \delta_\tau(e)$, which is a fixpoint of δ_τ by a well-known property of lco's.

Proposition 28 *Let $\delta(\tau)$ be an abstract garbage collector. We have that $\mathsf{fp}(\delta_\tau) = \mathsf{rng}(\alpha^{\mathcal{E}}_\tau)$ and $\varnothing \in \mathsf{fp}(\delta_\tau)$. Moreover, if `this` $\in \mathsf{dom}(\tau)$, then for every $X \subseteq \Sigma_\tau$ we have $\alpha^{\mathcal{E}}_\tau(X) = \varnothing$ if and only if $X = \varnothing$.*

Proposition 28 lets us assume that $\alpha^{\mathcal{E}}_\tau : \wp(\Sigma_\tau) \mapsto \mathsf{fp}(\delta_\tau)$. Moreover, it justifies the following definition of our domain $\mathcal{E}$ for escape analysis. Proposition 28 can be used to compute the possible approximations from $\mathcal{E}$ at a given program point. However, it does not specify which of these is best. This is the goal of an escape analysis (Subsection 4.2).

Definition 29 (Abstract Domain $\mathcal{E}$) *Our basic domain for escape analysis is $\mathcal{E}_\tau = \mathsf{fp}(\delta_\tau)$, ordered by set inclusion.*

Example 30 *See journal version of this paper.*

By Definition 22, we know that $\alpha^{\mathcal{E}}_\tau$ is strict and additive and, by Proposition 28, onto $\mathcal{E}_\tau$. Thus, by a general result of abstract interpretation [10, 11] (Section 2), we have the following proposition.

Proposition 31 *The map $\alpha^{\mathcal{E}}_\tau$ (Definition 22) is the abstraction map of a Galois insertion from $\wp(\Sigma_\tau)$ to $\mathcal{E}_\tau$.*

Note that if, in Definition 29, we had defined $\mathcal{E}_\tau$ as $\wp(\Pi)$, the map $\alpha^{\mathcal{E}}_\tau$ would induce just a Galois connection instead of a Galois insertion, as a consequence of Proposition 28.

The domain $\mathcal{E}$ induces optimal abstract operations which can be used for an actual escape analysis. We discuss this in the next subsection.

4.2. Static Analysis over $\mathcal{E}$

Figure 9 defines the abstract counterparts of the concrete operations in Figure 8. Proposition 32 states that they are correct and optimal, in the sense of abstract interpretation (Section 2). Optimality is proved by showing that each operation in Figure 9 coincides with the optimal operation $\alpha^{\mathcal{E}} \circ op \circ \gamma^{\mathcal{E}}$, where *op* is the corresponding concrete operation in Figure 8, as required by the abstract interpretation framework. Note that the map $\gamma^{\mathcal{E}}$ is induced by $\alpha^{\mathcal{E}}$ (Section 2).

Proposition 32 *The operations in Figure 9 are the optimal counterparts induced by $\alpha^{\mathcal{E}}$ of the operations in Figure 8 and of $\cup$. They are implicitly strict on $\varnothing$, except for* return*, which is strict in its first argument only, and for $\cup$.*

Many operations in Figure 9 coincide with the identity map. This is a sign of the computational imprecision conveyed by the domain $\mathcal{E}$. Other operations call the δ garbage collector quite often to remove creation points of objects which might become unreachable since some variable has disappeared from the scope. For instance, as the concrete $\mathsf{put\_var}$ operation removes variable v from the scope (Figure 8), its abstract counterpart in Figure 9 calls the garbage collector. The same happens for $\mathsf{restrict}$ which, however, removes a *set* of variables from the scope. There are also some operations ($\mathsf{is\_null}$, $\mathsf{put\_field}$, lookup) that use *res* as a temporary variable and one operation ($\mathsf{get\_field}$) that changes the type of *res*. Hence these abstract operations also need to call the garbage

$$
\begin{aligned}
\mathsf{nop}_\tau(e) &= e & \mathsf{get\_int}^{i}_\tau(e) &= e\\
\mathsf{get\_null}^{\kappa}_\tau(e) &= e & \mathsf{get\_var}^{v}_\tau(e) &= e\\
\mathsf{is\_true}_\tau(e) &= e & \mathsf{is\_false}_\tau(e) &= e\\
\mathsf{put\_var}^{v}_\tau(e) &= \delta_{\tau|_{-v}}(e) & \mathsf{is\_null}_\tau(e) &= \delta_{\tau|_{-res}}(e)\\
\mathsf{new}^{\pi}_\tau(e) &= e \cup \{\pi\} & =_\tau(e_1)(e_2) &= +_\tau(e_1)(e_2) = e_2\\
\mathsf{expand}^{v:t}_\tau(e) &= e & \mathsf{restrict}^{vs}_\tau(e) &= \delta_{\tau_{-vs}}(e)\\
\mathsf{call}^{\nu,v_1,\ldots,v_n}_\tau(e) &= \delta_{\tau|_{\{v_1,\ldots v_n,res\}}}(e) & \cup_\tau(e_1)(e_2) &= e_1 \cup e_2
\end{aligned}
$$

$$
\mathsf{get\_field}^{f}_\tau(e) = \begin{cases} \varnothing & \text{if } \{\pi \in e \mid k(\pi) \le \tau(res)\} = \varnothing\\ \delta_{\tau[res \mapsto F(\tau(res))(f)]}(e) & \text{otherwise}\end{cases}
$$

$$
\mathsf{put\_field}^{f}_{\tau,\tau'}(e_1)(e_2) = \begin{cases} \varnothing & \text{if } \{\pi \in e_1 \mid k(\pi) \le \tau(res)\} = \varnothing\\ \delta_{\tau|_{-res}}(e_2) & \text{otherwise}\end{cases}
$$

$$
\mathsf{return}^{\nu}_\tau(e_1)(e_2) = \cup \left\{ \{\pi\} \cup \delta_{F(k(\pi))}(\Pi) \;\middle|\; \begin{array}{l} \kappa \in \mathsf{rng}(\tau|_{-res}) \cap \mathcal{K}\\ \pi \in e_1,\ k(\pi) \le \kappa \end{array} \right\} \cup e_2
$$

$$
\mathsf{lookup}^{m,\nu}_\tau(e) = \begin{cases} \varnothing & \text{if } e' = \left\{ \pi \in e \;\middle|\; \begin{array}{l} k(\pi) \le \tau(res)\\ M(k(\pi))(m) = \nu \end{array} \right\} = \varnothing\\ \delta_{\tau|_{-res}}(e) \cup \left(\bigcup\{\{\pi\} \cup \delta_{F(k(\pi))}(e) \mid \pi \in e'\}\right) & \text{otherwise.}\end{cases}
$$

Figure 9. The optimal abstract operations over $\mathcal{E}$.

collector. Note that the definitions of the get_field, put_field and lookup operations also consider, separately, the unusual situation when we read a field, respectively, write a field or call a method and the receiver is *always null*. In this case, the concrete computation always stops so that the best approximation of the (empty) set of subsequent states is $\varnothing$. The garbage collector is also called by call since it creates a scope for the callee where only some of the variables of the caller (namely, the parameters of the callee) are addressable. The new operation adds its creation point to the approximation, since its concrete counterpart creates an object and binds it to the temporary variable *res*. The $\cup$ operation computes the union of the creation points reachable from at least one of the two branches of a conditional. The return operation states that all fields of the objects bound to the variables in scope before the call might have been modified by the call. This is reflected by the use of $\delta_{F(k(\pi))}(\Pi)$ in return, which plays the role of a worst-case assumption on the content of the fields. After Example 33 we discuss how to cope with the possible imprecision of this definition. The lookup operation computes first the set e' of the creation points of objects that may be receivers of the virtual call. If this set is not empty, the variable *res* (which holds the receiver of the call) is required to be bound to an object created at some creation point in e'. This further constrains the creation points reachable from *res* and this is why we call the garbage collector $\delta_{F(k(\pi))}$ for each $\pi \in e'$.

The definitions of return and lookup are quite complex; this is a consequence of our quest for *optimal* abstract operations. It is possible to replace their definitions in Figure 9 by the less precise but simpler definitions:

$$\mathsf{return}^{\nu}_{\tau}(e_1)(e_2) = \delta_\tau(\Pi) \cup e_2 \qquad \mathsf{lookup}^{m,\nu}_{\tau}(e) = e.$$

Note though that, in practice, the results with the simpler definitions will often be the same.

Example 33 *See journal version of this paper.*

There is, however, another problem related with the domain $\mathcal{E}$. It is exemplified below.

Example 34 *See journal version of this paper.*

5. The Refined Domain $\mathcal{ER}$

We define here a *refinement* $\mathcal{ER}$ of the domain $\mathcal{E}$ of Section 4, in the sense that $\mathcal{ER}$ is a concretisation of $\mathcal{E}$ (Proposition 56). The idea underlying the definition of $\mathcal{ER}$ is that the precision of $\mathcal{E}$ can be improved if we can speak about the creation points of the objects bound to *a given* variable or field (see the problem highlighted in Example 34). The construction of $\mathcal{ER}$ is very similar to that of $\mathcal{E}$.

5.1. The Domain

Definition 11 defines concrete values. The domain $\mathcal{ER}$ we are going to define approximates every concrete value with an *abstract value*. An abstract value is either $*$, which approximates the integers, or a set $e \subseteq \Pi$, which approximates *null* and all locations containing an object created in some creation point in e. An abstract frame maps variables to abstract values consistent with their type.

Definition 35 (Abstract Values and Frames) *Let the* abstract values *be* $Value^{\mathcal{ER}} = \{*\} \cup \wp(\Pi)$. *We define*

$$Frame^{\mathcal{ER}}_{\tau} = \left\{ \phi \in \mathsf{dom}(\tau) \mapsto Value^{\mathcal{ER}} \;\middle|\; \begin{array}{l} \textit{for every } v \in \mathsf{dom}(\tau) \\ \quad \textit{if } \tau(v) = \textit{int then } \phi(v) = * \\ \quad \textit{if } \tau(v) \in \mathcal{K} \textit{ and } \pi \in \phi(v) \\ \qquad \textit{then } k(\pi) \le \tau(v) \end{array} \right\}.$$

The set $Frame_\tau^{\mathcal{ER}}$ is ordered by pointwise set-inclusion.

EXAMPLE 36 *See journal version of this paper.*

The map ε *extracts* the creation points of the objects bound to the variables.

DEFINITION 37 (EXTRACTION MAP) *The map $\varepsilon_\tau : \wp(\Sigma_\tau) \mapsto Frame_\tau^{\mathcal{ER}}$ is such that, for every $S \subseteq \Sigma_\tau$ and $v \in \mathsf{dom}(\tau)$,*

$$\varepsilon_\tau(S)(v) = \begin{cases} * & \text{if } \tau(v) = int \\ \{(\mu\phi(v)).\pi \mid \phi \star \mu \in S \text{ and } \phi(v) \in Loc\} & \text{if } \tau(v) \in \mathcal{K}. \end{cases}$$

EXAMPLE 38 *See journal version of this paper.*

Since it is assumed that all the fields are uniquely identified by their fully qualified name, the type environment $\widetilde{\tau}$ *of all the fields* introduced by the program is well-defined.

DEFINITION 39 (TYPE ENVIRONMENT OF ALL FIELDS) *We define the* type environment of all fields *as $\widetilde{\tau} = \cup\{F(\kappa) \mid \kappa \in \mathcal{K}\}$. Let $\tau \in TypEnv$ be such that $\mathsf{dom}(\tau) \subseteq \mathsf{dom}(\widetilde{\tau})$ and $\phi \in Frame_\tau$. Its* extension *$\widetilde{\phi} \in Frame_{\widetilde{\tau}}$ is such that, for every $v \in \mathsf{dom}(\widetilde{\tau})$,*

$$\widetilde{\phi}(v) = \begin{cases} \phi(v) & \text{if } v \in \mathsf{dom}(\tau) \\ \Im(\widetilde{\tau}(v)) & \text{otherwise (Definition 11).} \end{cases}$$

EXAMPLE 40 *See journal version of this paper.* □

An abstract memory is an abstract frame for $\widetilde{\tau}$. The abstraction map computes the abstract memory by extracting the creation points of the fields of the reachable objects of the concrete memory (Definition 21).

DEFINITION 41 (ABSTRACT MAP FOR $\mathcal{ER}$) *Let the set of abstract memories be $Memory^{\mathcal{ER}} = Frame_{\widetilde{\tau}}^{\mathcal{ER}}$. We define the map*

$$\alpha_\tau^{\mathcal{ER}} : \wp(\Sigma_\tau) \mapsto \{\bot\} \cup (Frame_\tau^{\mathcal{ER}} \times Memory^{\mathcal{ER}})$$

such that, for $S \subseteq \Sigma_\tau$,

$$\alpha_\tau^{\mathcal{ER}}(S) = \begin{cases} \bot & \text{if } S = \varnothing \\ \varepsilon_\tau(S) \star \varepsilon_{\widetilde{\tau}}(\{\widetilde{o.\phi} \star \sigma.\mu \mid \sigma \in S \text{ and } o \in O_\tau(\sigma)\}) & \text{otherwise.} \end{cases}$$

EXAMPLE 42 *See journal version of this paper.*

Compare Examples 42 and 23. You can see that $\mathcal{ER}$ distributes over the variables and fields the same creation points observed by $\mathcal{E}$.

As a notational simplification, we often assume that each field not reported in the approximation of the memory is implicitly bound to $\varnothing$, if it has class type, and bound to $*$, if it has *int* type.

Just as for $\alpha^{\mathcal{E}}_{\tau}$ (Example 24), the following example shows that the map $\alpha^{\mathcal{ER}}_{\tau}$ is not necessarily onto.

EXAMPLE 43 *See journal version of this paper.*

Hence, we define a map ξ which forces to $\varnothing$ the fields of type class of the objects which have no reachable creation points. Just as for the garbage collector δ for $\mathcal{E}$, the map ξ can be seen as an abstract garbage collector for $\mathcal{ER}$. This ξ uses an auxiliary map ρ to compute the set of creation points r reachable from the variables in scope. The approximations of the fields of the objects created at r are not garbage collected by ξ. The approximations of the other fields are garbage collected instead.

DEFINITION 44 (ABSTRACT GARBAGE COLLECTOR ξ) *We define* $\rho_\tau : \mathit{Frame}^{\mathcal{ER}}_{\tau} \times \mathit{Memory}^{\mathcal{ER}} \mapsto \wp(\Pi)$ *and* $\xi_\tau : \{\bot\} \cup (\mathit{Frame}^{\mathcal{ER}}_{\tau} \times \mathit{Memory}^{\mathcal{ER}}) \mapsto \{\bot\} \cup (\mathit{Frame}^{\mathcal{ER}}_{\tau} \times \mathit{Memory}^{\mathcal{ER}})$ *as* $\rho_\tau(s) = \cup\{\rho^i_\tau(s) \mid i \geq 0\}$*, where*

$$\rho^0_\tau(\phi\star\mu) = \varnothing$$

$$\rho^{i+1}_\tau(\phi\star\mu) = \bigcup\left\{ \{\pi\} \cup \rho^i_{F(k(\pi))}(\mu|_{\mathsf{dom}(F(k(\pi)))}\star\mu) \,\middle|\, \begin{array}{l} v \in \mathsf{dom}(\tau) \\ \pi \in \phi(v) \end{array} \right\}$$

and

$$\xi_\tau(\bot) = \bot$$

$$\xi_\tau(\phi\star\mu) = \begin{cases} \bot & \text{if } \mathtt{this} \in \mathsf{dom}(\tau) \text{ and } \phi(\mathtt{this}) = \varnothing \\ \phi\star\left(\cup\{\mu|_{\mathsf{dom}(F(k(\pi)))} \mid \pi \in \rho_\tau(\phi\star\mu)\}\right) & \text{otherwise.} \end{cases}$$

EXAMPLE 45 *See journal version of this paper.*

The following property is expected to hold for a garbage collector. Compare Propositions 27 and 46.

PROPOSITION 46 *The abstract garbage collector* ξ_τ *is an lco.*

The garbage collector ξ_τ can be used to define $\mathsf{rng}(\alpha^{\mathcal{ER}}_{\tau})$. Namely, the *useful* elements of $\mathit{Frame}^{\mathcal{ER}}_{\tau} \times \mathit{Memory}^{\mathcal{ER}}$ are exactly those that do not contain any garbage. Compare Propositions 28 and 47.

PROPOSITION 47 *Let* ξ_τ *be the abstract garbage collector of Definition 44. Then* $\mathsf{fp}(\xi_\tau) = \mathsf{rng}(\alpha^{\mathcal{ER}}_{\tau})$.

Proposition 47 allows us to assume that $\alpha^{\mathcal{ER}}_\tau : \wp(\Sigma_\tau) \mapsto \mathsf{fp}(\xi_\tau)$ and justifies the following definition.

DEFINITION 48 (ABSTRACT DOMAIN $\mathcal{ER}$) *We define $\mathcal{ER}_\tau = \mathsf{fp}(\xi_\tau)$, ordered by pointwise set-inclusion (with the assumption that $* \subseteq *$ and $\perp \subseteq s$ for every $s \in \mathcal{ER}_\tau$).*

By Definitions 37 and 41 we know that the map $\alpha^{\mathcal{ER}}_\tau$ is strict and additive. By Proposition 47 we know that it is onto. Thus we have the following result corresponding to Proposition 31 for the domain $\mathcal{E}$.

PROPOSITION 49 *The map $\alpha^{\mathcal{ER}}_\tau$ is the abstraction map of a Galois insertion from $\wp(\Sigma_\tau)$ to $\mathcal{ER}_\tau$.*

5.2. STATIC ANALYSIS OVER $\mathcal{ER}$

In order to use the domain $\mathcal{ER}$ for an escape analysis, we need to provide the abstract counterparts over $\mathcal{ER}$ of the concrete operations in Figure 8. Since $\mathcal{ER}$ approximates every variable and field with an abstract value, those abstract operations are similar to those of the Palsberg and Schwartzbach's domain for *class analysis* in [23] as formulated in [31]. However, $\mathcal{ER}$ observes the fields of just the reachable objects (Definition 41), while Palsberg and Schwartzbach's domain observes the fields of all objects in memory.

Figure 10 reports the abstract counterparts on $\mathcal{ER}$ of the concrete operations in Figure 8. These operations are implicitly strict on $\perp$ except for $\cup$. In this case, we define $\perp \cup (\phi \star \mu) = (\phi \star \mu) \cup \perp = \phi \star \mu$. Their optimality is proved by showing that each operation in Figure 10 coincides with the optimal operation $\alpha^{\mathcal{ER}} \circ op \circ \gamma^{\mathcal{ER}}$, where *op* is the corresponding concrete operation in Figure 8, as required by the abstract interpretation framework. Note that the map $\gamma^{\mathcal{ER}}$ is induced by $\alpha^{\mathcal{ER}}$ (Section 2).

PROPOSITION 50 *The operations in Figure 10 are the optimal counterparts induced by $\alpha^{\mathcal{ER}}$ of the operations in Figure 8 and of $\cup$.*

Let us consider each of the abstract operations. The operation nop leaves the state unchanged. The same happens for the operations working with integer values only, such as $\mathsf{is\_true}$, $\mathsf{is\_false}$, $=$ and $+$, since the domain $\mathcal{ER}$ ignores variables with integer values. The concrete operation $\mathsf{get\_int}$ loads an integer into *res*. Hence, its abstract counterpart loads $*$ into *res*, since $*$ is the approximation for integer values (Definition 35). The concrete operation $\mathsf{get\_null}$ loads *null* into *res* and

$$
\begin{aligned}
\mathsf{nop}_\tau(\phi \star \mu) &= \phi \star \mu \\
\mathsf{get\_int}^i_\tau(\phi \star \mu) &= \phi[\mathit{res} \mapsto *] \star \mu \\
\mathsf{get\_null}^\kappa_\tau(\phi \star \mu) &= \phi[\mathit{res} \mapsto \varnothing] \star \mu \\
\mathsf{get\_var}^v_\tau(\phi \star \mu) &= \phi[\mathit{res} \mapsto \phi(v)] \star \mu \\
\mathsf{is\_true}_\tau(\phi \star \mu) &= \phi \star \mu \\
\mathsf{is\_false}_\tau(\phi \star \mu) &= \phi \star \mu \\
\cup_\tau(\phi_1 \star \mu_1)(\phi_2 \star \mu_2) &= (\phi_1 \cup \phi_2) \star (\mu_1 \cup \mu_2) \\
\mathsf{is\_null}_\tau(\phi \star \mu) &= \xi_{\tau[\mathit{res} \mapsto \mathit{int}]}(\phi[\mathit{res} \mapsto *] \star \mu) \\
\mathsf{new}^\pi_\tau(\phi \star \mu) &= \phi[\mathit{res} \mapsto \{\pi\}] \star \mu \\
\mathsf{put\_var}^v_\tau(\phi \star \mu) &= \xi_{\tau|_{-\mathit{res}}}(\phi[v \mapsto \phi(\mathit{res})]|_{-\mathit{res}} \star \mu) \\
\mathsf{restrict}^{vs}_\tau(\phi \star \mu) &= \xi_{\tau|_{-vs}}(\phi|_{-vs} \star \mu) \\
\mathsf{expand}^{v:t}_\tau(\phi \star \mu) &= \begin{cases} \phi[v \mapsto *] \star \mu & \text{if } t = \mathit{int} \\ \phi[v \mapsto \varnothing] \star \mu & \text{otherwise} \end{cases} \\
=_\tau(\phi_1 \star \mu_1)(\phi_2 \star \mu_2) &= +_\tau(\phi_1 \star \mu_1)(\phi_2 \star \mu_2) = \phi_2 \star \mu_2
\end{aligned}
$$

$$
\mathsf{get\_field}^f_\tau(\phi \star \mu) = \begin{cases} \bot & \text{if } \phi(\mathit{res}) = \varnothing \\ \xi_{\tau[\mathit{res} \mapsto F(\tau(\mathit{res}))(f)]}(\phi[\mathit{res} \mapsto \mu(f)] \star \mu) & \text{else} \end{cases}
$$

$$
\begin{array}{c} \mathsf{put\_field}^f_{\tau,\tau'} \\ (\phi_1 \star \mu_1)(\phi_2 \star \mu_2) \end{array} = \begin{cases} \bot \quad \text{if } \phi_1(\mathit{res}) = \varnothing \\ \xi_{\tau|_{-\mathit{res}}}(\phi_2|_{-\mathit{res}} \star \mu_2) \\ \quad \text{else, if no } \pi \in \phi_1(\mathit{res}) \text{ occurs in } \phi_2|_{-\mathit{res}} \star \mu_2 \\ \xi_{\tau|_{-\mathit{res}}}(\phi_2|_{-\mathit{res}} \star \mu_2[f \mapsto \mu_2(f) \cup \phi_2(\mathit{res})]) \\ \quad \text{otherwise} \end{cases}
$$

$$
\mathsf{call}^{\nu, v_1, \ldots, v_n}_\tau(\phi \star \mu) = \xi_{P(\nu)|_{-\mathtt{out}}}\left(\begin{bmatrix} \iota_1 \mapsto \phi(v_1), \ldots, \iota_n \mapsto \phi(v_n) \\ \mathtt{this} \mapsto \phi(\mathit{res}) \end{bmatrix} \star \mu\right)
$$

$$
\begin{array}{c} \mathsf{return}^\nu_\tau \\ (\phi_1 \star \mu_1)(\phi_2 \star \mu_2) \end{array} = \xi_{\tau|_{-\mathit{res}}}(\phi_1|_{-\mathit{res}} \star \mu^\top) \cup ([\mathit{res} \mapsto \phi_2(\mathtt{out})] \star \mu_2)
$$

$$
\text{where } \mu^\top \text{ is the top of } \mathit{Memory}^{\mathcal{ER}}
$$

$$
\mathsf{lookup}^{m,\nu}_\tau(\phi \star \mu) = \begin{cases} \bot & \text{if } e = \{\pi \in \phi(\mathit{res}) \mid M(\pi)(m) = \nu\} = \varnothing \\ \xi_\tau(\phi[\mathit{res} \mapsto e] \star \mu) & \text{otherwise.} \end{cases}
$$

Figure 10. The abstract operations over $\mathcal{ER}$.

hence its abstract counterpart approximates *res* with $\varnothing$. The operation $\mathsf{get\_var}^v$ copies the creation points of v into those of *res*. The $\cup$ operation merges the creation points of the objects bound to each given variable or field in one of the two branches of a conditional. The concrete $\mathsf{is\_null}$ operation checks if *res* contains *null* or not, and loads 1 or -1 in *res* accordingly. Hence its abstract counterpart loads $*$ into *res*. Since the old value of *res* may no longer be reachable, we apply the abstract garbage collector ξ. The new^{π} operation binds *res* to an object created at π. The $\mathsf{put\_var}^{v}$ operation copies the value of *res* into v, and removes *res*. Since the old value of v may be lost, we apply the abstract garbage collector ξ. The restrict operation removes some variables from the scope and, hence, calls ξ. The expand^{v} operation adds the variable v in scope. Its initial value is approximated with $*$, if it is 0, and with $\varnothing$, if it is *null*. The $\mathsf{get\_field}^{f}$ operation returns $\perp$ if it is *always* applied to states where the receiver *res* is *null*. This is because $\perp$ is the best approximation of the empty set of final states. If, instead, the receiver is not necessarily *null*, the creation points of the field f are copied from the approximation $\mu(f)$ into the approximation of *res*. Since this operation changes the value of *res*, possibly making some object unreachable, it needs to call ξ. For the $\mathsf{put\_field}^{f}$ operation, we first check if the receiver is always *null*, in which case the abstract operation returns $\perp$. Then we consider the case in which the evaluation of what is going to be put inside the field makes the receiver unreachable. This (pathological) case happens in a situation such as `a.g.f` $=$ `m(a)` where the method call `m(a)` sets to *null* the field `g` of the object bound to `a`. Since we assume that the left-hand side is evaluated before the right-hand side, the receiver is not necessarily *null*, but the field updates might not be observable if `a.g.f` is only reachable from `a`. In the third and final case for put_field we consider the standard situation when we write into a reachable field of a non-*null* receiver. The creation points of the right-hand side are added to those already approximating the objects stored in f. The call operation restricts the scope to the parameters passed to a method and hence ξ is used. The return operation copies into *res* the return value of the method which is held in `out`. The local variables of the caller are put back into scope, but the approximation of their fields is provided through a worst-case assumption $\mu^{\top}$ since they may be modified by the call. This loss of precision can be overcome by means of shadow copies of the variables, just as for $\mathcal{E}$ (see Example 52). The lookup^{m} operation first computes the subset e of the approximation of the receiver of the call only containing the creation points whose class leads to a call to the method m. If $e = \varnothing$, a call to m is impossible and the result of the operation is $\perp$. Otherwise, e becomes the approximation of the receiver *res*, so that some creation points can disappear and we need to call ξ.

EXAMPLE 51 *See journal version of this paper.*

The abstract state s''_6 shows that the imprecision problem of $\mathcal{E}$, related to the return operation, is still present in $\mathcal{ER}$. By comparing

s_2 with s''_6, it can be seen that the `return` operation makes a very pessimistic assumption about the possible creation points for the `next` and `rotation` fields. In particular, from s''_6 it seems that creation points π_3 and π_4 are reachable (they belong to $\mu^\top$), which is not the case in the concrete state (compare this with σ''_6 in Example 19). As for the domain $\mathcal{E}$, this problem can be solved by including, in the state of the callee, *shadow copies* of the parameters of the caller. This is implemented through a preprocessing of the bodies of the methods which prepend statements of the form $v'{:=}v$ for each parameter v, where v' is the shadow copy of v. Since shadow copies are fresh new variables, not already occurring in the method's body, their value is never changed. In this way, at the end of the method we know which creation points are reachable from the fields of the objects bound to such parameters.

EXAMPLE 52 *See journal version of this paper.*

As previously noted in Subsection 1.2, shadow copies of the parameters are also useful for dealing with methods that modify their formal parameters.

There was another problem with $\mathcal{E}$, related to the fact that $\mathcal{E}$ does not distinguish between different variables (see end of Section 4). It is not surprising that $\mathcal{ER}$ solves that problem, as shown below.

EXAMPLE 53 *See journal version of this paper.*

5.3. $\mathcal{ER}$ IS A REFINEMENT OF $\mathcal{E}$

We have called $\mathcal{ER}$ a *refinement* of $\mathcal{E}$. In order to give this word a formal justification, we show here that $\mathcal{ER}$ actually includes the elements of $\mathcal{E}$. Namely, we show how every element $e \in \mathcal{E}$ can be *embedded* into an element $\theta(e)$ of $\mathcal{ER}$, such that e and $\theta(e)$ have the same concretisation *i.e.,* they represent the same property of concrete states. The idea, formalised in Definition 54, is that every variable or field must be bound in $\mathcal{ER}$ to all those creation points in e compatible with its type.

DEFINITION 54 (EMBEDDING OF $\mathcal{E}$ INTO $\mathcal{ER}$) *Let* $s \subseteq \Pi$. *We define* $\vartheta_\tau(s) \in \mathit{Frame}^{\mathcal{ER}}_\tau$ *such that, for every* $v \in \mathsf{dom}(\tau)$,

$$\vartheta_\tau(s)(v) = \begin{cases} * & \text{if } \tau(v) = \mathit{int} \\ \{\pi \in s \mid k(\pi) \le \tau(v)\} & \text{if } \tau(v) \in \mathcal{K}. \end{cases}$$

The embedding $\theta_\tau(e) \in \mathcal{ER}_\tau$ *of* $e \in \mathcal{E}_\tau$ *is* $\theta_\tau(e) = \xi_\tau(\vartheta_\tau(e) \star \vartheta_{\widetilde{\tau}}(e))$.

EXAMPLE 55 *See journal version of this paper.*

Proposition 56 states that the embedding of Definition 54 is correct. The proof proceeds by showing that $\theta_\tau(e)$ is an element of $\mathcal{ER}_\tau$ and approximates exactly the same concrete states as e, that is, for every element of $\mathcal{E}$ there is an element of $\mathcal{ER}$ which represents exactly the same set of concrete states.

PROPOSITION 56 *Let $\gamma_\tau^{\mathcal{E}}$ and $\gamma_\tau^{\mathcal{ER}}$ be the concretisation maps induced by the abstraction maps of Definitions 22 and 41, respectively. Then $\gamma_\tau^{\mathcal{E}}(\mathcal{E}_\tau) \subseteq \gamma_\tau^{\mathcal{ER}}(\mathcal{ER}_\tau)$.*

The following example shows that the inclusion relation in Proposition 56 must be strict.

EXAMPLE 57 *See journal version of this paper.*

6. Implementation

See journal version of this paper.

7. Discussion

8. Conclusion

We have presented a formal development of an escape analysis by abstract interpretation, providing optimality results in the form of a Galois insertion from the concrete to the abstract domain and of the definition of optimal abstract operations. This escape analysis has been implemented and applied to full Java (bytecode). This results in an escape analyser which is probably less precise than others already developed, but still performs well in practice from the points of view of its cost and precision .

A first, basic escape domain $\mathcal{E}$ is defined as a property of concrete states (Definition 29). This domain is simple but non-trivial since

- The set of the creation points of the objects reachable from the current state can both grow (new) and shrink (δ); *i.e., static type information contains escape information* (Examples 24 and 33);
- That set is useful, sometimes, to restrict the possible targets of a virtual call *i.e., escape information contains class information* (Example 33).

However, the escape analysis induced by our domain $\mathcal{E}$ is not precise enough from a computational point of view, since it induces rather imprecise abstract operations. We have therefore defined a refinement $\mathcal{ER}$ of $\mathcal{E}$, on the basis of the information that $\mathcal{E}$ lacks, in order to attain better precision. The relation between $\mathcal{ER}$ and $\mathcal{E}$ is similar to that between Palsberg and Schwartzbach's class analysis [23, 31] and *rapid type analysis* [3] although, while all objects stored in memory are considered in [3, 31, 23], only those actually reachable from the variables in scope are considered by the domains $\mathcal{E}$ and $\mathcal{ER}$ (Definitions 22 and 41). The ability to describe only the reachable objects, through the use of an abstract garbage collector (δ in Figure 9 and ξ in Figure 10), improves the precision of the analysis, since it becomes focused on only those objects that can actually affect the concrete execution of the program.

It is interesting to consider if this notion of reachability and the use of an abstract garbage collector can be applied to other static analyses of the run-time heap as well. Namely, class, shape, sharing and cyclicity analyses might benefit from them.

Acknowledgements

This work has been funded by the Italian MURST grant *Abstract Interpretation, Type Systems and Control-Flow Analysis* and by the British EPSRC grant GR/R53401.

References

1. G. Agrawal. Simultaneous Demand-Driven Data-flow and Call Graph Analysis. In *Proc. of the International Conference on Software Maintenance (ICSM'99)*, pages 453–462, Oxford, UK, September 1999. IEEE Computer Society.
2. K. Arnold, J. Gosling, and D. Holmes. *The Java^{TM} Programming Language*. Addison-Wesley, third edition, 2000.
3. D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proc. of OOPSLA'96*, volume 31(10) of *ACM SIGPLAN Notices*, pages 324–341, New York, 1996. ACM Press.
4. B. Blanchet. Escape Analysis: Correctness Proof, Implementation and Experimental Results. In *25th ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages (POPL'98)*, pages 25–37, San Diego, CA, USA, January 1998. ACM Press.
5. B. Blanchet. Escape Analysis for Java: Theory and Practice. *ACM TOPLAS*, 25(6):713–775, November 2003.
6. J. Bogda and U. Hölzle. Removing Unnecessary Synchronization in Java. In *Proc. of OOPSLA'99*, volume 34(10) of *SIGPLAN Notices*, pages 35–46, Denver, Colorado, USA, November 1999.

7. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-Semantics Approach: Theory and Applications. *Journal of Logic Programming*, 19/20:149–197, 1994.
8. J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis. *ACM TOPLAS*, 25(6):876–910, November 2003.
9. A. Cortesi, G. Filé, and W. Winsborough. The Quotient of an Abstract Interpretation. *Theoretical Computer Science*, 202(1-2):163–192, 1998.
10. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.
11. P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.
12. D. R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, The Netherlands, July 1996.
13. A. Deutsch. On the Complexity of Escape Analysis. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 358–371, Paris, France, January 1997. ACM Press.
14. D. Gay and B. Steensgaard. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In D. A. Watt, editor, *Compiler Construction, 9th International Conference (CC'00)*, volume 1781 of *Lecture Notes in Computer Science*, pages 82–93. Springer-Verlag, Berlin, March 2000.
15. R. Giacobazzi and F. Ranzato. Refining and Compressing Abstract Domains. In *Proc. of the 24th International Colloquium on Automata, Languages and Programming (ICALP'97)*, volume 1256 of *LNCS*, pages 771–781. Springer-Verlag, 1997.
16. R. Giacobazzi and F. Scozzari. A Logical Model for Relational Abstract Domains. *ACM Transactions on Programming Languages and Systems*, 20(5):1067–1109, 1998.
17. P. M. Hill and F. Spoto. A Foundation of Escape Analysis. In H. Kirchner and C. Ringeissen, editors, *Proc. of AMAST'02*, volume 2422 of *LNCS*, pages 380–395, St. Gilles les Bains, La Réunion island, France, September 2002. Springer-Verlag.
18. P. M. Hill and F. Spoto. A Refinement of the Escape Property. In A. Cortesi, editor, *Proc. of the VMCAI'02 workshop on Verification, Model-Checking and Abstract Interpretation*, volume 2294 of *Lecture Notes in Computer Science*, pages 154–166, Venice, Italy, January 2002. Springer-Verlag.
19. T. Jensen. Disjunctive Program Analysis for Algebraic Data Types. *ACM Transactions on Programming Languages and Systems*, 19(5):752–804, 1997.
20. N. D. Jones and H. Søndergaard. A Semantics-based Framework for the Abstract Interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood Ltd, 1987.
21. T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, second edition, 1999.
22. Cousot. P. and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. of the Sixth Annual ACM Symposium on Principles of Programming Languages (POPL'79)*, pages 269–282, San Antonio, Texas, 1979. ACM.
23. J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proc. of OOPSLA'91*, volume 26(11) of *ACM SIGPLAN Notices*, pages 146–161. ACM Press, November 1991.

24. Y. G. Park and B. Goldberg. Escape Analysis on Lists. In *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92)*, volume 27(7) of *SIGPLAN Notices*, pages 116–127, San Francisco, California, USA, June 1992.
25. A. Rountev, A. Milanova, and B. G. Ryder. Points-to Analysis for Java Using Annotated Constraints. In *Proc. of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'01)*, volume 36(11) of *ACM SIGPLAN*, pages 43–55, Tampa, Florida, USA, October 2001.
26. E. Ruf. Effective Synchronization Removal for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, volume 35(5) of *SIGPLAN Notices*, pages 208–218, Vancouver, British Columbia, Canada, June 2000.
27. C. Ruggieri and T. P. Murtagh. Lifetime Analysis of Dynamically Allocated Objects. In *15th ACM Symposium on Principles of Programming Languages (POPL'88)*, pages 285–293, San Diego, California, USA, January 1988.
28. A. Salcianu and M. Rinard. Pointer and Escape Analysis for Multithreaded Programs. In *Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*, volume 36(7) of *SIGPLAN Notices*, pages 12–23, Snowbird, Utah, USA, July 2001.
29. F. Scozzari. Logical Optimality of Groundness Analysis. *Theoretical Computer Science*, 277(1-2):149–184, 2002.
30. H. Søndergaard. An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction. In B. Robinet and R. Wilhelm, editors, *Proc. of the European Symposium on Programming (ESOP)*, volume 213 of *Lecture Notes in Computer Science*, pages 327–338, Saarbrücken, Federal Republic of Germany, March 1986. Springer.
31. F. Spoto and T. Jensen. Class Analyses as Abstract Interpretations of Trace Semantics. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(5):578–630, September 2003.
32. M. Streckenbach and G. Snelting. Points-to for Java: A General Framework and an Empirical Comparison. Technical report, Universität Passau, Germany, November 2000.
33. A. Tarski. A Lattice-theoretical Fixpoint Theorem and its Applications. *Pacific J. Math.*, 5:285–309, 1955.
34. F. Vivien and M. Rinard. Incrementalized Pointer and Escape Analysis. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, volume 36(5) of *SIGPLAN Notices*, pages 35–46, Snowbird, Utah, USA, June 2001.
35. J. Whaley and M. S. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In W. Pugh and C. Chambers, editors, *Proc. of ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04)*, pages 131–144, Washington, DC, USA, June 2004. ACM.
36. J. Whaley and M. C. Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 34(1) of *SIGPLAN Notices*, pages 187–206, Denver, Colorado, USA, November 1999.
37. G. Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, 1993.

Appendix

A. Proofs of Propositions 27, 28 and 32 in Section 4.

Proposition 27. *Let $i \in \mathbb{N}$. The abstract garbage collectors δ_τ^i and δ_τ are lco's.*

Proof. Since $\delta_\tau = \delta_\tau^{\#\Pi}$, it is enough to prove the result for δ_τ^i only. By Definition 25, the maps δ_τ^i for $i \in \mathbb{N}$ are reductive and monotonic. We prove idempotency by induction over $i \in \mathbb{N}$. Let $e \subseteq \Pi$. We have $\delta_\tau^0 \delta_\tau^0(e) = \delta_\tau^0(\varnothing) = \varnothing = \delta_\tau^0(e)$. Assume that the result holds for a given $i \in \mathbb{N}$. If $\mathtt{this} \in \mathsf{dom}(\tau)$ and there is no $\pi \in e$ such that $k(\pi) \leq \tau(\mathtt{this})$, then $\delta_\tau^i \delta_\tau^i(e) = \delta_\tau^i(\varnothing) = \varnothing = \delta_\tau^i(e)$. Suppose now that, if $\mathtt{this} \in \mathsf{dom}(\tau)$, then there exists $\pi \in e$ such that $k(\pi) \leq \tau(\mathtt{this})$. By reductivity, $\delta_\tau^{i+1}\delta_\tau^{i+1}(e) \subseteq \delta_\tau^{i+1}(e)$. We prove that the converse inclusion holds. We have

$$\delta_\tau^{i+1}\delta_\tau^{i+1}(e) = \cup \left\{ \{\pi\} \cup \delta^i_{F(k(\pi))} \delta_\tau^{i+1}(e) \;\middle|\; \begin{array}{l} \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K} \\ \pi \in \delta_\tau^{i+1}(e),\ k(\pi) \leq \kappa \end{array} \right\}. \tag{1}$$

Let $\kappa \in \mathsf{rng}(\tau) \cap \mathcal{K}$ and $\pi \in \Pi$ be such that $k(\pi) \leq \kappa$. If $\pi \in \delta_\tau^{i+1}(e)$ then, by reductivity, we have $\pi \in e$. Conversely, if $\pi \in e$ then, by Definition 25, $\pi \in \delta_\tau^{i+1}(e)$. We conclude from (1) that

$$\begin{aligned}
\delta_\tau^{i+1}\delta_\tau^{i+1}(e) &= \cup \left\{ \{\pi\} \cup \delta^i_{F(k(\pi))} \delta_\tau^{i+1}(e) \;\middle|\; \begin{array}{l} \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K} \\ \pi \in e,\ k(\pi) \leq \kappa \end{array} \right\} \\
\text{(monotonicity)} &\supseteq \cup \left\{ \{\pi\} \cup \delta^i_{F(k(\pi))} \delta^i_{F(\pi)}(e) \;\middle|\; \begin{array}{l} \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K} \\ \pi \in e,\ k(\pi) \leq \kappa \end{array} \right\} \\
\text{(ind. hypothesis)} &= \cup \left\{ \{\pi\} \cup \delta^i_{F(k(\pi))}(e) \;\middle|\; \begin{array}{l} \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K} \\ \pi \in e,\ k(\pi) \leq \kappa \end{array} \right\} \\
&= \delta_\tau^{i+1}(e).
\end{aligned}$$

□

To prove Proposition 28, we need some preliminary definitions and results. We start by defining, for every $i \in \mathbb{N}$, a map α_τ^i which, for sufficiently large i, coincides with $\alpha_\tau^{\mathcal{E}}$ (Definition 22).

Definition 58 *Let $i \in \mathbb{N}$. We define the map $\alpha_\tau^i : \wp(\Sigma_\tau) \mapsto \Pi$ as $\alpha_\tau^i(S) = \{o.\pi \mid \sigma \in S$ and $o \in O_\tau^i(\sigma)\}$ (see Definition 21 for O_τ^i).*

Corollary 59 *Let $S \subseteq \Sigma_\tau$ and $i \geq 0$. We have*

$$\alpha_\tau^{i+1}(S) = \bigcup \left\{ \{o.\pi\} \cup \alpha^i_{F(k(o.\pi))}(o.\phi \star \mu) \;\middle|\; \begin{array}{l} \phi \star \mu \in \Sigma_\tau,\ v \in \mathrm{dom}(\tau) \\ \phi(v) \in \mathit{Loc},\ o = \mu\phi(v) \end{array} \right\} .$$

Proof. By Definitions 58 and 21. □

Lemma 60 states that α^i_τ (and hence also $\alpha^{\mathcal{E}}_\tau$ itself) yields sets of creation points that do not contain garbage.

Lemma 60 *Let $\sigma \in \Sigma_\tau$ and $i \in \mathbb{N}$. Then $\alpha^i_\tau(\sigma) = \delta^i_\tau \alpha^i_\tau(\sigma)$.*

Proof. By reductivity (Proposition 27), we have $\alpha^i_\tau(\sigma) \supseteq \delta^i_\tau \alpha^i_\tau(\sigma)$. It remains to prove $\alpha^i_\tau(\sigma) \subseteq \delta^i_\tau \alpha^i_\tau(\sigma)$. Let $\sigma = \phi \star \mu$. We proceed by induction on i. We have $\alpha^0_\tau(\sigma) = \varnothing = \delta^0_\tau \alpha^0_\tau(\sigma)$. Assume that the property holds for a given $i \in \mathbb{N}$. Let $\tau' = F(k(o.\pi))$ and $X = \{\mu\phi(v) \mid v \in \mathrm{dom}(\phi) \text{ and } \phi(v) \in \mathit{Loc}\}$. By Corollary 59,

$$\begin{aligned} \alpha^{i+1}_\tau(\sigma) &= \cup\{\{o.\pi\} \cup \alpha^i_{\tau'}(o.\phi \star \mu) \mid o \in X\} \\ \text{(inductive hypothesis)} &= \cup\{\{o.\pi\} \cup \delta^i_{\tau'} \alpha^i_{\tau'}(o.\phi \star \mu) \mid o \in X\} \ . \end{aligned} \tag{2}$$

By Corollary 59, we have $\alpha^i_{\tau'}(o.\phi \star \mu) \subseteq \alpha^{i+1}_\tau(\sigma)$ and, by Proposition 27, (2) is contained in

$$\cup\{\{o.\pi\} \cup \delta^i_{\tau'} \alpha^{i+1}_\tau(\sigma) \mid o \in X\} \ . \tag{3}$$

Note that, given $o \in X$, we can always find $\kappa \in \mathsf{rng}(\tau) \cap \mathcal{K}$ such that $k(o.\pi) \le \kappa$. Indeed, for the definition of X, there exists $v \in \mathrm{dom}(\phi) = \mathrm{dom}(\tau)$ such that $\phi(v) \in \mathit{Loc}$ and $o = \mu\phi(v)$. By Definition 11, we have $\tau(v) \in \mathcal{K}$. By Definition 14, we have $k(o.\pi) = k((\mu\phi(v)).\pi) \le \tau(v)$. Hence letting $\kappa = \tau(v)$, (3) is

$$\begin{aligned} & \cup \left\{ \{o.\pi\} \cup \delta^i_{\tau'} \alpha^{i+1}_\tau(\sigma) \,\middle|\, \begin{array}{l} o \in X,\ \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K} \\ k(o.\pi) \le \kappa \end{array} \right\} \\ \text{(Corollary 59)} \subseteq & \cup \left\{ \{\pi\} \cup \delta^i_{\tau'} \alpha^{i+1}_\tau(\sigma) \,\middle|\, \begin{array}{l} \pi \in \alpha^{i+1}_\tau(\sigma),\ \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K} \\ k(\pi) \le \kappa \end{array} \right\} \\ \text{(Definition 25)} = & \ \delta^{i+1}_\tau \alpha^{i+1}_\tau(\sigma) \ . \end{aligned}$$

Note that the last step is correct since if $\mathtt{this} \in \mathrm{dom}(\tau)$ we have $\phi(\mathtt{this}) \ne \mathit{null}$ (Definition 16). Hence $(\mu\phi(\mathtt{this})).\pi \in \alpha^{i+1}_\tau(\sigma)$ and $k((\mu\phi(\mathtt{this})).\pi) \le \tau(\mathtt{this})$ (Definition 13). We conclude that, if $\mathtt{this} \in \mathrm{dom}(\tau)$, then there exists $\pi \in \alpha^{i+1}_\tau(\sigma)$ such that $k(\pi) \le \tau(\mathtt{this})$. □

Let e be a set of creation points. We now define frames and memories which use all possible creation points in e allowed by the type environment of the variables. In this sense, they are the *richest* frames and memories containing creation points from e only.

Definition 61 *Let $\{\pi_1, \dots, \pi_n\}$ be an enumeration without repetitions of Π. Let $l_1, \dots, l_n$ be distinct locations. Let $e \subseteq \Pi$ and $w \in \mathrm{dom}(\tau)$ such that $\tau(w) \in \mathcal{K}$. We define*

$$L_\tau(e, w) = \{l_i \mid 1 \le i \le n,\ \pi_i \in e \textit{ and } k(\pi_i) \le \tau(w)\}\ ,$$

$$\overline{\phi}_\tau(e) = \left\{ \phi \in \textit{Frame}_\tau \;\middle|\; \begin{array}{l} \textit{for every } v \in \mathrm{dom}(\tau) \\ \ \tau(v) = \textit{int} \Rightarrow \phi(v) = 0 \\ \ \tau(v) \in \mathcal{K},\ L_\tau(e, v) = \varnothing \Rightarrow \phi(v) = \textit{null} \\ \ \tau(v) \in \mathcal{K},\ L_\tau(e, v) \ne \varnothing \Rightarrow \phi(v) \in L_\tau(e, v) \end{array} \right\},$$

$$\overline{\mu}(e) = \left\{ \mu \in \textit{Memory} \;\middle|\; \begin{array}{l} \mu = [l_1 \mapsto \pi_1 \star \phi_1, \dots, l_n \mapsto \pi_n \star \phi_n] \\ \textit{and } \phi_i \in \overline{\phi}_{F(k(\pi_i))}(e) \textit{ for } i = 1, \dots, n \end{array} \right\}\ .$$

We prove now some properties of the frames and memories of Definition 61.

Lemma 62 *Let $e_1, e_2 \subseteq \Pi$, $\phi \in \overline{\phi}_\tau(e_1)$ and $\mu \in \overline{\mu}(e_2)$. Then*

i) $\phi \star \mu : \tau$;

ii) $\phi \star \mu \in \Sigma_\tau$ iff $\mathtt{this} \notin \mathrm{dom}(\tau)$ or there exists $\pi \in e_1$ s.t. $k(\pi) \le \tau(\mathtt{this})$;

iii) If $\phi \star \mu \in \Sigma_\tau$ then $\alpha_\tau(\phi \star \mu) \subseteq e_1 \cup e_2$.

Proof.

i) Condition 1 of Definition 14 is satisfied since we have that $\mathsf{rng}(\phi) \cap \textit{Loc} \subseteq \{l_1, \dots, l_n\} = \mathrm{dom}(\mu)$. Moreover, if $v \in \mathrm{dom}(\phi)$ and $\phi(v) \in \textit{Loc}$ then $\phi(v) \in L_\tau(e_1, v)$. Thus there exists $1 \le i \le n$ such that $\phi(v) = l_i$, $(\mu\phi(v)).\pi = \pi_i$ and $k((\mu\phi(v)).\pi) = k(\pi_i) \le \tau(v)$. Condition 2 of Definition 14 holds because if $o \in \mathsf{rng}(\mu)$ then $o.\phi = \phi_i$ for some $1 \le i \le n$. Since $\phi_i \in \overline{\phi}_{F(k(\pi_i))}(e)$, reasoning as above we conclude that ϕ_i is $F(k(\pi_i))$-correct *w.r.t.* μ. Then $\phi \star \mu : \tau$.

ii) By point i, we know that $\phi \star \mu : \tau$. From Definition 16, we have $\phi \star \mu \in \Sigma_\tau$ if and only if $\mathtt{this} \notin \mathrm{dom}(\tau)$ or $\phi(\mathtt{this}) \ne \textit{null}$. By Definition 61, the latter case holds if and only if $L_\tau(e_1, \mathtt{this}) \ne \varnothing$ *i.e.,* if and only if there exists $\pi \in e_1$ such that $k(\pi) \le \tau(\mathtt{this})$.

iii) Since $\phi \star \mu \in \Sigma_\tau$, the α_τ map is defined (Definition 22). Let

$$L = (\mathsf{rng}(\phi) \cup (\cup\{\mathsf{rng}(o.\phi) \mid o \in \mathsf{rng}(\mu)\})) \cap \textit{Loc}\ .$$

Since $\phi \in \overline{\phi}_\tau(e_1)$ and $o.\phi \in \overline{\phi}_{F(k(o.\pi))}(e_2)$ for every $o \in \mathsf{rng}(\mu)$, by Definition 61, we have

$$\{\mu(l).\pi \mid l \in L\} \subseteq e_1 \cup e_2\ .$$

By Definition 22, we conclude that

$$\alpha_\tau(\phi \star \mu) \subseteq \{\mu(l).\pi \mid l \in L\} \subseteq e_1 \cup e_2 \ .$$

□

Lemma 63 gives an explicit definition of the abstraction of the set of states constructed from the frames and memories of Definition 61.

LEMMA 63 *Let $e_1, e_2 \subseteq \Pi$, $j \in \mathbb{N}$ and*

$$A^j = \alpha_\tau^{j+1}(\{\phi \star \mu \in \Sigma_\tau \mid \phi \in \overline{\phi}_\tau(e_1) \textit{ and } \mu \in \overline{\mu}(e_2)\}) \ .$$

Then

$$A^j = \begin{cases} \varnothing & \textit{if } \texttt{this} \in \mathrm{dom}(\tau) \textit{ and there is no } \pi \in e_1 \textit{ s.t. } k(\pi) \leq \tau(\texttt{this}) \\ \cup \left\{ \{\pi\} \cup \delta^j_{F(k(\pi))}(e_2) \,\middle|\, \begin{array}{l} v \in \mathrm{dom}(\tau),\ \tau(v) \in \mathcal{K} \\ \pi \in e_1,\ k(\pi) \leq \tau(v) \end{array} \right\} & \textit{otherwise.} \end{cases}$$

Proof. We proceed by induction over j. By Lemma 62.ii, if $j = 0$ we have

$$A^0 = \begin{cases} \varnothing & \text{if } \texttt{this} \in \mathrm{dom}(\tau) \text{ and there is no } \pi \in e_1 \text{ s.t. } k(\pi) \leq \tau(\texttt{this}) \\ \left\{ o.\pi \,\middle|\, \begin{array}{l} \phi \in \overline{\phi}_\tau(e_1),\ \mu \in \overline{\mu}(e_2),\ v \in \mathrm{dom}(\phi) \\ \phi(v) \in \mathit{Loc},\ o = \mu\phi(v) \end{array} \right\} & \text{otherwise.} \end{cases}$$

By Definition 61, the latter case is equal to

$$\left\{ \pi_i \,\middle|\, \begin{array}{l} v \in \mathrm{dom}(\tau),\ \tau(v) \in \mathcal{K} \\ 1 \leq i \leq n,\ \pi_i \in e_1 \\ k(\pi_i) \leq \tau(v) \end{array} \right\} = \cup \left\{ \{\pi\} \cup \delta^0_{F(k(\pi))}(e_2) \,\middle|\, \begin{array}{l} v \in \mathrm{dom}(\tau) \\ \tau(v) \in \mathcal{K} \\ \pi \in e_1 \\ k(\pi) \leq \tau(v) \end{array} \right\} .$$

Assume now that the result holds for a given $j \in \mathbb{N}$. If $\texttt{this} \in \mathrm{dom}(\tau)$ and there is no $\pi \in e_1$ such that $k(\pi) \leq \tau(\texttt{this})$, by Lemma 62.ii, we have $A^{j+1} = \varnothing$. Otherwise, by Corollary 59 we have

$$A^{j+1} = \cup \left\{ \{o.\pi\} \cup \alpha^{j+1}_{F(k(o.\pi))}(o.\phi \star \mu) \,\middle|\, \begin{array}{l} \phi \in \overline{\phi}_\tau(e_1),\ \mu \in \overline{\mu}(e_2) \\ v \in \mathrm{dom}(\phi) \\ \phi(v) \in \mathit{Loc},\ o = \mu\phi(v) \end{array} \right\} . \quad (4)$$

As for the base case, we know that $o.\pi$ ranges over $\{\pi \in e_1 \mid v \in \mathrm{dom}(\tau),\ \tau(v) \in \mathcal{K},\ k(\pi) \leq \tau(v)\}$. Since $o.\phi \in \overline{\phi}_{F(k(o.\pi))}(e_2)$ is arbitrary (Definition 61), by the inductive hypothesis, (4) becomes

$$\cup \left\{ \{\pi\} \cup \alpha^{j+1}_{F(k(\pi))}\left(\left\{ \phi \star \mu \;\middle|\; \begin{array}{l} \phi \in \overline{\phi}_{F(k(\pi))}(e_2) \\ \mu \in \overline{\mu}(e_2) \end{array} \right\}\right) \;\middle|\; \begin{array}{l} v \in \mathrm{dom}(\tau) \\ \tau(v) \in \mathcal{K} \\ \pi \in e_1 \\ k(\pi) \leq \tau(v) \end{array} \right\}$$

$$= \cup \left\{ \{\pi\} \cup \delta^{j+1}_{F(k(\pi))}(e_2) \mid v \in \mathrm{dom}(\tau),\ \tau(v) \in \mathcal{K},\ \pi \in e_1,\ k(\pi) \leq \tau(v) \right\}.$$

□

COROLLARY 64 *Let* $e_1, e_2 \subseteq \Pi$. *Let*

$$A_\tau(e_1, e_2) = \alpha_\tau(\{\phi \star \mu \in \Sigma_\tau \mid \phi \in \overline{\phi}_\tau(e_1) \textit{ and } \mu \in \overline{\mu}(e_2)\}) \ .$$

Then

i) $$A_\tau(e_1, e_2) = \begin{cases} \varnothing & \textit{if } \texttt{this} \in \mathrm{dom}(\tau) \\ & \quad \textit{and no } \pi \in e_1 \textit{ is s.t. } k(\pi) \leq \tau(\texttt{this}) \\ \cup \left\{ \{\pi\} \cup \delta_{F(k(\pi))}(e_2) \;\middle|\; \begin{array}{l} v \in \mathrm{dom}(\phi),\ \tau(v) \in \mathcal{K} \\ \pi \in e_1,\ k(\pi) \leq \tau(v) \end{array} \right\} \\ \quad \textit{otherwise,} \end{cases}$$

ii) $A_\tau(e_1, e_1) = \delta_\tau(e_1)$.

Proof. Point i follows by Lemma 63 since j is arbitrary. Point ii follows from point i and Definition 25. □

COROLLARY 65 *Let* $\kappa \in \mathcal{K}$, $\tau = [res \mapsto \kappa]$, p *be a predicate over* Π *and* $e \subseteq \Pi$ *be such that there exists* $\pi \in e$ *such that* $k(\pi) \leq \tau(res)$ *and* $p(\pi)$ *holds. Then*

$$\begin{aligned} &\alpha_\tau(\{\phi \star \mu \in \Sigma_\tau \mid \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e),\ p(\mu\phi(res).\pi)\}) \\ &\qquad\qquad = \cup\{\{\pi\} \cup \delta_{F(k(\pi))}(e) \mid \pi \in e,\ k(\pi) \leq \tau(res),\ p(\pi)\} \ . \end{aligned}$$

Proof. Let $j \in \mathbb{N}$. By the hypothesis on e and Corollary 59 we have

$$\begin{aligned} &\alpha^{j+1}_\tau(\{\phi \star \mu \in \Sigma_\tau \mid \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e),\ p(\mu\phi(res).\pi)\}) \\ &= \cup \left\{ \{o.\pi\} \cup \alpha^j_{F(k(o.\pi))}(o.\phi \star \mu) \;\middle|\; \begin{array}{l} \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e) \\ o = \mu\phi(res),\ p(o.\pi) \end{array} \right\} \\ &= \cup \left\{ \{\pi\} \cup \alpha^j_{F(k(\pi))}(\phi' \star \mu) \;\middle|\; \begin{array}{l} \pi \in e,\ k(\pi) \leq \tau(res),\ p(\pi) \\ \phi' \in \overline{\phi}_{F(k(\pi))}(e),\ \mu \in \overline{\mu}(e) \end{array} \right\}. \end{aligned}$$

Since j is arbitrary we have

$$\alpha_\tau(\{\phi\star\mu \in \Sigma_\tau \mid \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e),\ p(\mu\phi(res).\pi)\})$$
$$= \cup\left\{\{\pi\} \cup \alpha_{F(k(\pi))}\left(\left\{\phi'\star\mu \,\middle|\, \begin{array}{l}\phi' \in \overline{\phi}_{F(k(\pi))}(e)\\ \mu \in \overline{\mu}(e)\end{array}\right\}\right) \,\middle|\, \begin{array}{l}\pi \in e,\ p(\pi)\\ k(\pi) \leq \tau(res)\end{array}\right\},$$

and the thesis follows by Corollary 64.ii. □

Proposition 28. *Let $\delta(\tau)$ be an abstract garbage collector. Then we have $\mathsf{fp}(\delta_\tau) = \mathsf{rng}(\alpha_\tau^{\mathcal{E}})$ and $\varnothing \in \mathsf{fp}(\delta_\tau)$. Moreover, if* `this` *$\in \mathsf{dom}(\tau)$, then for every $X \subseteq \Sigma_\tau$ we have $\alpha_\tau^{\mathcal{E}}(X) = \varnothing$ if and only if $X = \varnothing$.*

Proof. We first prove that $\mathsf{fp}(\delta_\tau) = \mathsf{rng}(\alpha_\tau)$. Let $X \subseteq \Sigma_\tau$ and $i \in \mathbb{N}$. By Lemma 60 and monotonicity (Proposition 27) we have

$$\begin{aligned}\alpha_\tau^i(X) &= \cup\{\alpha_\tau^i(\sigma) \mid \sigma \in X\}\\ &= \cup\{\delta_\tau^i\alpha_\tau^i(\sigma) \mid \sigma \in X\} \subseteq \delta_\tau^i\alpha_\tau^i(X) \subseteq \delta_\tau\alpha_\tau^i(X)\ .\end{aligned}$$

The converse inclusion $\alpha_\tau^i(X) \subseteq \delta_\tau\alpha_\tau^i(X)$ holds because δ_τ is reductive (Proposition 27). Then $\alpha_\tau^i(X) \in \mathsf{fp}(\delta_\tau)$. Since i is arbitrary we have $\alpha_\tau(X) \in \mathsf{fp}(\delta_\tau)$. Conversely, let $e \in \mathsf{fp}(\delta_\tau)$. Consider the set of states constructed from the frames and memories in Definition 61 and let

$$X = \{\phi\star\mu \in \Sigma_\tau \mid \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e)\}\ .$$

By Corollary 64.ii and since $e \in \mathsf{fp}(\delta_\tau)$, we have $\alpha_\tau(X) = \delta_\tau(e) = e$.

Since δ_τ is reductive (Proposition 27), we have $\varnothing = \delta_\tau(\varnothing)$ *i.e.,* $\varnothing \in \mathsf{fp}(\delta_\tau)$.

If `this` $\in \mathrm{dom}(\tau)$, every $\sigma \in \Sigma_\tau$ is such that $\alpha_\tau(\sigma) \neq \varnothing$, since `this` cannot be unbound (Definition 16). Then $\alpha_\tau(X) = \varnothing$ if and only if $X = \varnothing$.

The proof of Proposition 32 requires some preliminary results.

Corollary 66 states that if we know that the approximation of a set of concrete states S is some $e \subseteq \Pi$, then we can conclude that a better approximation of S is $\delta(e)$. In other words, garbage is never used in the approximation.

Corollary 66 *Let $S \subseteq \Sigma_\tau$ and $e \subseteq \Pi$. Then $\alpha_\tau(S) \subseteq \delta_\tau(e)$ if and only if $\alpha_\tau(S) \subseteq e$.*

Proof. Assume that $\alpha_\tau(S) \subseteq \delta_\tau(e)$. By reductivity (Proposition 27) we have $\alpha_\tau(S) \subseteq e$. Conversely, assume that $\alpha_\tau(S) \subseteq e$. By Proposition 28 and monotonicity (Proposition 27) we have $\alpha_\tau(S) = \delta_\tau\alpha_\tau(S) \subseteq \delta_\tau(e)$. □

Lemma 67 states that integer values, *null* and the name of the variables are not relevant to the definition of α (Definition 22).

LEMMA 67 *Let $\phi' \star \mu \in \Sigma_{\tau'}$ and $\phi'' \star \mu \in \Sigma_{\tau''}$ such that $\mathsf{rng}(\phi') \cap Loc = \mathsf{rng}(\phi'') \cap Loc$. Then $\alpha_{\tau'}(\phi' \star \mu) = \alpha_{\tau''}(\phi'' \star \mu)$.*

Proof. From Definition 22. □

Lemma 68 says that if we consider all the concrete states approximated by some $e \subseteq \Pi$ and we restrict their frames, then the resulting set of states is approximated by $\delta(e)$. In other words, the operation δ garbage collects all objects that, because of the restriction, are not longer reachable.

LEMMA 68 *Let $vs \subseteq \mathrm{dom}(\tau)$. Then*

$$\alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \star \mu \mid \phi \star \mu \in \Sigma_\tau \textit{ and } \alpha_\tau(\phi \star \mu) \subseteq e\}) = \delta_{\tau|_{-vs}}(e) \ .$$

Proof. We have

$$\begin{aligned} &\alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq e\}) \\ &= \alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \star \mu \in \Sigma_{\tau|_{-vs}} \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq e\}) \ , \qquad (5) \end{aligned}$$

since if $\phi \star \mu \in \Sigma_\tau$ then $\phi|_{-vs} \star \mu \in \Sigma_{\tau|_{-vs}}$. We have that if $\alpha_\tau(\phi \star \mu) \subseteq e$ then $\alpha_{\tau|_{-vs}}(\phi|_{-vs} \star \mu) \subseteq e$. Hence (5) is contained in e. By Corollary 66, (5) is also contained in $\delta_{\tau|_{-vs}}(e)$. But also the converse inclusion holds, since in (5) we can restrict the choice of $\phi \star \mu \in \Sigma_\tau$, so that (5) contains

$$\alpha_{\tau|_{-vs}}\left(\left\{\phi|_{-vs} \star \mu \in \Sigma_{\tau|_{-vs}} \;\middle|\; \begin{array}{l} \phi \star \mu \in \Sigma_\tau,\ \alpha_\tau(\phi \star \mu) \subseteq e \\ \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e) \end{array}\right\}\right) \ . \qquad (6)$$

By points ii and iii of Lemma 62, (6) is equal to

$$\begin{aligned} &\alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \star \mu \in \Sigma_{\tau|_{-vs}} \mid \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e)\}) \\ \text{(Definition 61)} &= \alpha_{\tau|_{-vs}}(\{\phi \star \mu \in \Sigma_{\tau|_{-vs}} \mid \phi \in \overline{\phi}_{\tau|_{-vs}}(e) \text{ and } \mu \in \overline{\mu}(e)\}) \\ \text{(Corollary 64.ii)} &= \delta_{\tau|_{-vs}}(e) \ . \end{aligned}$$

□

We are now ready to prove the correctness and optimality of the abstract operations in Figure 9.

Proposition 31. *The map $\alpha^{\mathcal{E}}_\tau$ (Definition 22) is the abstraction map of a Galois insertion from $\wp(\Sigma_\tau)$ to $\mathcal{E}_\tau$.*

Proof. By the theory of abstract interpretation [10], given $e \in \mathcal{E}_\tau$, the concretisation map induced by the abstraction map of Definition 22 is

$$\gamma_\tau(e) = \{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq e\} \ .$$

Moreover, the optimal abstract counterpart of a concrete operation op is $\alpha op \gamma$.

We consider every operation in Figure 8 and we compute the induced optimal abstract operation, which will always coincide with that reported in Figure 9.

Note that all the operations in Figure 8 use states in Σ_τ with $\texttt{this} \in \mathrm{dom}(\tau)$ (Figure 7). By Proposition 28 we have $\gamma_\tau(\varnothing) = \varnothing$. Then the powerset extension of the operations in Figure 8 are strict on $\varnothing$. The only exception is the second argument of return, which is a state whose frame is not required to contain $\texttt{this}$ (Figure 7). The operation $\cup$ is not the powerset extension of an operation in Figure 8. Then it is not strict in general. Hence, in the following, we will consider just the cases when the arguments of the abstract counterparts of the operations in Figure 8 are not $\varnothing$ (except for the second argument of return and for $\cup$).

In this proof, we will use the following properties.

P1 If $e \in \mathcal{E}_\tau$, $e \neq \varnothing$ and $\texttt{this} \in \mathrm{dom}(\tau)$ then there exists $\pi \in e$ such that $k(\pi) \leq \tau(\texttt{this})$.

P2 If $e \in \mathcal{E}_\tau$, $e \neq \varnothing$ and $\texttt{this} \in \mathrm{dom}(\tau)$ then there exists $\sigma \in \Sigma_\tau$ such that $\alpha_\tau(\sigma) \subseteq e$.

P3 $\alpha_\tau\gamma_\tau$ is the identity map.

P1 holds since $e = \delta_\tau(e)$ (Definition 29) so that by Definition 25, we can conclude that there exists such a π. To see that P2 is a consequence of P1, let π be as defined in P1; then, letting $\sigma = [\texttt{this} \mapsto l] \star [l \mapsto \pi \star \Im(F(k(\pi)))]$ for some $l \in \mathit{Loc}$, we have $\sigma \in \Sigma_\tau$. Moreover, by Definition 22, $\alpha_\tau(\sigma) = \{\pi\} \subseteq e$ so that P2 holds. By Proposition 31, α_τ is a Galois insertion and hence, P3 holds.

$\underline{\mathsf{nop}}$
By P3 we have

$$\alpha_\tau(\mathsf{nop}_\tau(\gamma_\tau(e))) = \alpha_\tau\gamma_\tau(e) = e \ .$$

<u>get_int, get_null, get_var</u>

$$\begin{aligned}
&\alpha_{\tau[res\mapsto int]}(\mathsf{get\_int}^i_\tau(\gamma_\tau(e)))\\
&= \alpha_{\tau[res\mapsto int]}(\{\phi[res\mapsto i]\star\mu \mid \phi\star\mu\in\gamma_\tau(e)\})\\
(*)\ &= \alpha_\tau(\{\phi\star\mu\in\Sigma_\tau \mid \phi\star\mu\in\gamma_\tau(e)\}) = \alpha_\tau\gamma_\tau(e) = e\ ,
\end{aligned}$$

where $*$ follows by Lemma 67 since $res \notin \mathrm{dom}(\tau)$. For the same reason, point $*$ follows if res is bound to $null$ or to some $\phi(v)$ with $v \in \mathrm{dom}(\tau)$. Thus the proof above is also a proof of the optimality of get_null and of get_var.

<u>expand</u>

$$\begin{aligned}
&\alpha_{\tau[v\mapsto t]}(\mathsf{expand}^{v:t}_\tau(\gamma_\tau(e)))\\
&= \alpha_{\tau[v\mapsto t]}(\{\phi[v\mapsto \Im(t)]\star\mu \mid \phi\star\mu\in\gamma_\tau(e)\})\\
(*)\ &= \alpha_\tau(\{\phi\star\mu\in\Sigma_\tau \mid \phi\star\mu\in\gamma_\tau(e)\}) = \alpha_\tau\gamma_\tau(e) = e\ ,
\end{aligned}$$

where point $*$ follows by Lemma 67, since $\Im(t) \in \{0, null\}$ and $v \notin \mathrm{dom}(\tau)$.

<u>restrict</u>

$$\begin{aligned}
&\alpha_{\tau|_{-vs}}(\mathsf{restrict}^{vs}_\tau(\gamma_\tau(e)))\\
&= \alpha_{\tau|_{-vs}}(\mathsf{restrict}^{vs}_\tau(\{\sigma\in\Sigma_\tau \mid \alpha_\tau(\sigma)\subseteq e\}))\\
&= \alpha_{\tau|_{-vs}}(\{\phi|_{-vs}\star\mu \mid \phi\star\mu\in\Sigma_\tau \text{ and } \alpha_\tau(\phi\star\mu)\subseteq e\})\\
(\text{Lemma 68})\ &= \delta_{\tau|_{-vs}}(e)\ .
\end{aligned}$$

<u>is_null</u>

$$\begin{aligned}
&\alpha_{\tau[res\mapsto int]}(\mathsf{is\_null}_\tau(\gamma_\tau(e)))\\
&= \alpha_{\tau[res\mapsto int]}(\mathsf{is\_null}_\tau(\{\sigma\in\Sigma_\tau \mid \alpha_\tau(\sigma)\subseteq e\}))\\
&= \alpha_{\tau[res\mapsto int]}\left(\left\{\phi[res\mapsto 1]\star\mu \,\middle|\, \begin{array}{l}\phi\star\mu\in\Sigma_\tau\\ \alpha_\tau(\phi\star\mu)\subseteq e\end{array}\right\}\right)\\
(\text{Lemma 67})\ &= \alpha_{\tau|_{-res}}(\{\phi|_{-res}\star\mu \mid \phi\star\mu\in\Sigma_\tau \text{ and } \alpha_\tau(\phi\star\mu)\subseteq e\})\\
(\text{Lemma 68})\ &= \delta_{\tau|_{-res}}(e)\\
(\text{Definition 25})\ &= \delta_{\tau[res\mapsto int]}(e)\ .
\end{aligned}$$

<u>put_var</u>

$$\begin{aligned}
&\alpha_{\tau|_{-res}}(\mathsf{put\_var}_\tau(\gamma_\tau(e)))\\
&= \alpha_{\tau|_{-res}}(\mathsf{put\_var}_\tau(\{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq e\}))\\
&= \alpha_{\tau|_{-res}}(\{\phi[v \mapsto \phi(res)]|_{-res} \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq e\}) \ .
\end{aligned} \tag{7}$$

Observe that $\mathsf{rng}(\phi[v \mapsto \phi(res)]|_{-res}) = \mathsf{rng}(\phi|_{-v})$ so that, by Lemmas 67 and 68, (7) is equal to

$$\alpha_{\tau|_{-v}}(\{\phi|_{-v} \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq e\}) = \delta_{\tau|_{-v}}(e) \ .$$

<u>call</u>

$$\begin{aligned}
&\alpha_{P(\nu)|_{-\mathsf{out}}}(\mathsf{call}_\tau^{\nu,v_1,\ldots,v_n}(\gamma_\tau(e)))\\
&= \alpha_{P(\nu)|_{-\mathsf{out}}}(\mathsf{call}_\tau^{\nu,v_1,\ldots,v_n}(\{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq e\}))\\
&= \alpha_{P(\nu)|_{-\mathsf{out}}}\left(\left\{ \begin{bmatrix} \iota_1 \mapsto \phi(v_1), \\ \vdots \\ \iota_n \mapsto \phi(v_n), \\ \mathtt{this} \mapsto \phi(res) \end{bmatrix} \star \mu \;\middle|\; \begin{array}{l} \phi \star \mu \in \Sigma_\tau \text{ and} \\ \alpha_\tau(\phi \star \mu) \subseteq e \end{array} \right\}\right)\\
(*) &= \alpha_{\tau|_{\{v_1,\ldots,v_n,res\}}}(\{\phi|_{\{v_1,\ldots,v_n,res\}} \star \mu \mid \phi \star \mu \in \Sigma_\tau,\ \alpha_\tau(\phi \star \mu) \subseteq e\})\\
(**) &= \delta_{\tau|_{\{v_1,\ldots,v_n,res\}}}(e) \ ,
\end{aligned}$$

where point $*$ follows by Lemma 67 and point $**$ follows by Lemma 68.

<u>is_true, is_false</u>

$$\begin{aligned}
&\alpha_\tau(\mathsf{is\_true}_\tau(\gamma_\tau(e)))\\
&= \alpha_\tau(\{\phi \star \mu \in \gamma_\tau(e) \mid \phi(res) \geq 0\})\\
(\text{Lemma 67}) &= \alpha_\tau \gamma_\tau(e) = e \ .
\end{aligned}$$

The optimality of is_false follows by a similar proof.

<u>new</u>

Let $\kappa = k(\pi)$. Since $res \notin \mathrm{dom}(\tau)$ we have

$$\begin{aligned}
&\alpha_{\tau[res\mapsto\kappa]}(\mathsf{new}^{\pi}_{\tau}(\gamma_\tau(e)))\\
&= \alpha_{\tau[res\mapsto\kappa]}(\mathsf{new}^{\pi}_{\tau}(\{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq e\}))\\
&= \alpha_{\tau[res\mapsto\kappa]}\left(\left\{\begin{array}{c} \phi[res \mapsto l]\star \\ \star\,\mu[l \mapsto \pi \star \Im(F(\kappa))] \end{array} \middle| \begin{array}{l} \phi\star\mu \in \Sigma_\tau,\ \alpha_\tau(\phi\star\mu) \subseteq e \\ l \in Loc \setminus \mathrm{dom}(\mu) \end{array}\right\}\right)\\
&= \alpha_\tau(\{\phi\star\mu \in \Sigma_\tau \mid \alpha_\tau(\phi\star\mu) \subseteq e\}) \cup \qquad (8)\\
&\quad \cup\, \alpha_{[res\mapsto\kappa]}\left(\left\{\begin{array}{c} [res \mapsto l]\star \\ \star[l \mapsto \pi \star \Im(F(\kappa))] \end{array} \middle| \begin{array}{l} \phi\star\mu \in \Sigma_\tau,\ \alpha_\tau(\phi\star\mu) \subseteq e \\ l \in Loc \setminus \mathrm{dom}(\mu) \end{array}\right\}\right). \qquad (9)
\end{aligned}$$

We have that (8) is equal to e. By P2 and Definition 22, (9) is equal to $\{\pi\}$.

<u>$=$, $+$</u>

$$\begin{aligned}
&\alpha_\tau(=_\tau(\gamma_\tau(e_1))(\gamma_\tau(e_2)))\\
&= \alpha_\tau(\{=_\tau(\sigma_1)(\sigma_2) \mid \sigma_1 \in \gamma_\tau(e_1),\ \sigma_2 \in \gamma_\tau(e_2)\})\\
(\mathrm{P2}) &= \alpha_\tau(\{\sigma_2 \mid \sigma_2 \in \gamma_\tau(e_2)\})\\
&= \alpha_\tau\gamma_\tau(e_2) = e_2\ .
\end{aligned}$$

The optimality of $+$ follows by a similar proof.

<u>return</u>
Let $\tau' = \tau[res \mapsto P(\nu)(\mathtt{out})]$, $\tau'' = P(\nu)|_{\mathtt{out}}$ and $L = \mathsf{rng}(\phi_1|_{-res}) \cap Loc$.

$$\begin{aligned}
&\alpha_{\tau'}(\mathsf{return}^{\nu}_{\tau}(\gamma_\tau(e_1))(\gamma_{\tau''}(e_2)))\\
&= \alpha_{\tau'}(\mathsf{return}^{\nu}_{\tau}(\{\sigma_1 \in \Sigma_\tau \mid \alpha_\tau(\sigma_1) \subseteq e_1\})(\{\sigma_2 \in \Sigma_{\tau''} \mid \alpha_{\tau''}(\sigma_2) \subseteq e_2\}))\\
&= \alpha_{\tau'}\left(\left\{ \phi_1|_{-res}[res \mapsto \phi_2(\mathtt{out})]\star\mu_2 \middle| \underbrace{\begin{array}{c} \phi_1\star\mu_1 \in \Sigma_\tau \\ \phi_2\star\mu_2 \in \Sigma_{\tau''} \\ \alpha_\tau(\phi_1\star\mu_1) \subseteq e_1 \\ \alpha_{\tau''}(\phi_2\star\mu_2) \subseteq e_2 \\ \mu_1 =_L \mu_2 \end{array}}_{Cond} \right\}\right)\\
(*) &= \alpha_{\tau|_{-res}}(\{\phi_1|_{-res}\star\mu_2 \mid Cond\}) \cup \qquad (10)\\
&\quad \cup\, \alpha_{\tau''}(\{\phi_2\star\mu_2 \mid Cond\}) \qquad (11)
\end{aligned}$$

where point $*$ follows by Lemma 67. Since $\alpha_{\tau''}(\phi_2\star\mu_2) \subseteq e_2$, an upper bound of (11) is e_2. But e_2 is also a lower bound of (11) since, by

Lemma 62.iii, a lower bound of (11) is

$$\alpha_{\tau''}\left(\left\{\phi_2 \star \mu_2 \;\middle|\; \begin{array}{l}\phi_1 \in \overline{\phi}_{\tau}(e_1),\ \mu_1 \in \overline{\mu}(e_1)\\ \phi_2 \in \overline{\phi}_{\tau''}(e_2),\ \mu_2 \in \overline{\mu}(e_2)\end{array}\right\}\right)$$

which by Corollary 64.ii is equal to e_2. Note that the condition $\mu_1 =_L \mu_2$ is satisfied by Definition 61.

Instead (10) is

$$\cup \left\{ \{o.\pi\} \cup \alpha_{F(k(o.\pi))}(o.\phi \star \mu_2) \;\middle|\; \begin{array}{l} v \in \mathrm{dom}(\phi_1|_{-res})\\ \phi_1|_{-res}(v) \in Loc\\ o = \mu_2\phi_1|_{-res}(v),\ Cond \end{array}\right\}$$

which, since $\mu_1 =_L \mu_2$, is equal to

$$\begin{array}{rl} & \cup \left\{ \{o.\pi\} \cup \alpha_{F(k(o.\pi))}(o.\phi \star \mu_2) \;\middle|\; \begin{array}{l} v \in \mathrm{dom}(\phi_1|_{-res})\\ \phi_1|_{-res}(v) \in Loc\\ o = \mu_1\phi_1|_{-res}(v),\ Cond \end{array}\right\}\\ (*) \subseteq & \cup \left\{ \{o.\pi\} \cup \delta_{F(k(o.\pi))}(\Pi) \;\middle|\; \begin{array}{l} v \in \mathrm{dom}(\phi_1|_{-res}),\ \phi_1|_{-res}(v) \in Loc\\ o = \mu_1\phi_1|_{-res}(v),\ Cond \end{array}\right\}\\ (**) \subseteq & \cup \left\{ \{\pi\} \cup \delta_{F(k(\pi))}(\Pi) \;\middle|\; \begin{array}{l} \kappa \in \mathsf{rng}(\tau|_{-res}) \cap \mathcal{K}\\ \pi \in e_1,\ k(\pi) \leq \kappa,\ Cond \end{array}\right\}\\ \subseteq & \cup \left\{ \{\pi\} \cup \delta_{F(k(\pi))}(\Pi) \;\middle|\; \begin{array}{l} \kappa \in \mathsf{rng}(\tau|_{-res}) \cap \mathcal{K}\\ \pi \in e_1,\ k(\pi) \leq \kappa \end{array}\right\}\ , \end{array} \qquad (12)$$

where point $*$ follows by Lemma 60 and point $**$ holds since $Cond$ requires that $\alpha_\tau(\phi_1 \star \mu_1) \subseteq e_1$. But (12) is also a lower bound of (10), since (10) contains

$$\begin{array}{l} \alpha_{\tau|_{-res}}\left(\left\{ \phi_1|_{-res} \star \mu_2 \in \Sigma_{\tau|_{-res}} \;\middle|\; \begin{array}{l} \phi_1 \in \overline{\phi}_\tau(e_1),\ \mu_1 \in \overline{\mu}(e_1),\\ \phi_2 = \Im(P(\nu)|_{\mathsf{out}}),\ \mu_2 \in \overline{\mu}(\Pi) \end{array}\right\}\right)\\ \qquad = \alpha_{\tau|_{-res}}(\{\phi \star \mu \in \Sigma_{\tau|_{-res}} \mid \phi \in \overline{\phi}_{\tau|_{-res}}(e_1), \mu \in \overline{\mu}(\Pi)\})\ , \end{array}$$

which by Corollary 64.i is equal to (12).

<u>get_field</u>
Let $\tau' = \tau[res \mapsto F(\tau(res))(f)]$ and $\tau'' = [res \mapsto F(\tau(res))(f)]$. We

have

$$\begin{aligned}
&\alpha_{\tau'}(\mathsf{get\_field}^f_\tau(\gamma_\tau(e)))\\
&= \alpha_{\tau'}(\mathsf{get\_field}^f_\tau(\{\phi \star \mu \in \Sigma_\tau \mid \alpha_\tau(\phi \star \mu) \subseteq e\}))\\
&= \alpha_{\tau'}\left(\left\{ \phi|_{-res}[res \mapsto (\mu\phi(res)).\phi(f)] \star \mu \;\middle|\; \begin{array}{l} \phi \star \mu \in \Sigma_\tau \\ \phi(res) \neq null, \\ \alpha_\tau(\phi \star \mu) \subseteq e \end{array} \right\}\right)\\
&= \alpha_{\tau|_{-res}}(\{\phi|_{-res} \star \mu \mid \phi \star \mu \in \Sigma_\tau,\ \phi(res) \neq null,\ \alpha_\tau(\phi \star \mu) \subseteq e\}) \cup\\
&\quad \cup\, \alpha_{\tau''}\left(\left\{ [res \mapsto (\mu\phi(res)).\phi(f)] \star \mu \;\middle|\; \begin{array}{l} \phi \star \mu \in \Sigma_\tau \\ \phi(res) \neq null, \\ \alpha_\tau(\phi \star \mu) \subseteq e \end{array} \right\}\right) . \qquad (13)
\end{aligned}$$

(13) is equal to $\varnothing$ if $\{\pi \in e \mid k(\pi) \leq \tau(res)\} = \varnothing$, since in such a case the condition $\phi(res) \neq null$ cannot be satisfied. Since $\alpha_\tau(\phi \star \mu) \subseteq e$, an upper bound of (13) is e. By Corollary 66, also $\delta_{\tau'}(e)$ is an upper bound of (13). But it is also a lower bound of (13), since, from the hypothesis on e and from points ii and iii of Lemma 62, (13) contains

$$\begin{aligned}
&\alpha_{\tau|_{-res}}(\{\phi|_{-res} \star \mu \in \Sigma_{\tau|_{-res}} \mid \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e)\}) \cup\\
&\cup\, \alpha_{\tau''}(\{[res \mapsto (\mu\phi(res)).\phi(f)] \star \mu \in \Sigma_{\tau''} \mid \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e)\})\\
(*) &= \alpha_{\tau|_{-res}}(\{\phi \star \mu \in \Sigma_{\tau|_{-res}} \mid \phi \in \overline{\phi}_{\tau|_{-res}}(e),\ \mu \in \overline{\mu}(e)\}) \cup\\
&\cup\, \alpha_{\tau''}(\{\phi \star \mu \in \Sigma_{\tau''} \mid \phi \in \overline{\phi}_{\tau''}(e),\ \mu \in \overline{\mu}(e)\})\\
(**) &= \delta_{\tau|_{-res}}(e) \cup \delta_{\tau''}(e) = \delta_{\tau'}(e) ,
\end{aligned}$$

where point $*$ follows by Definition 61 and point $**$ follows by Corollary 64.ii.

<u>lookup</u>

$$\begin{aligned}
&\alpha_\tau(\mathsf{lookup}^{m,\nu}_\tau(\gamma_\tau(e)))\\
&= \alpha_\tau(\mathsf{lookup}^{m,\nu}_\tau(\{\phi \star \mu \in \Sigma_\tau \mid \alpha_\tau(\phi \star \mu) \subseteq e\}))\\
&= \alpha_\tau\left(\left\{ \phi \star \mu \in \Sigma_\tau \;\middle|\; \underbrace{\begin{array}{l} \alpha_\tau(\phi \star \mu) \subseteq e,\ \phi(res) \neq null \\ M(k((\mu\phi(res)).\pi))(m) = \nu \end{array}}_{Cond} \right\}\right) . \qquad (14)
\end{aligned}$$

Equation (14) is equal to $\varnothing$ if there is no $\pi \in e$ such that $k(\pi) \leq \tau(res)$ and $M(\pi)(m) = \nu$, because in such a case it is not possible to satisfy the condition $M(k((\mu\phi(res)).\pi))(m) = \nu$. Otherwise, it is equal to

$$\alpha_{\tau|_{-res}}(\{\phi|_{-res} \star \mu \mid \phi \star \mu \in \Sigma_\tau,\ Cond\}) \cup \qquad (15)$$

$$\cup\, \alpha_{\tau|_{res}}(\{\phi|_{res} \star \mu \mid \phi \star \mu \in \Sigma_\tau,\ Cond\}) . \qquad (16)$$

Since $Cond$ requires that $\alpha_\tau(\phi \star \mu) \subseteq e$, by Corollary 66 an upper bound of (15) is $\delta_{\tau|_{-res}}(e)$. But it is also a lower bound of (15), since a lower bound of (15) is

$$\alpha_{\tau|_{-res}}\left(\left\{\phi|_{-res} \star \mu \in \Sigma_{\tau|_{-res}} \;\middle|\; \begin{array}{l} \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e) \\ \phi(res) \neq null \\ M(k((\mu\phi(res)).\pi))(m) = \nu \end{array}\right\}\right)$$
$$(*) = \alpha_{\tau|_{-res}}(\{\phi \star \mu \in \Sigma_{\tau|_{-res}} \mid \phi \in \overline{\phi}_{\tau|_{-res}}(e),\ \mu \in \overline{\mu}(e)\})$$
$$(**) = \delta_{\tau|_{-res}}(e)\ .$$

Point $*$ follows from the hypothesis on e. Point $**$ follows by Corollary 64.ii.

Instead, (16) is contained in

$$\alpha_{\tau|_{res}}\left(\left\{\phi|_{res} \star \mu \in \Sigma_{\tau|_{res}} \;\middle|\; \begin{array}{l} \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e) \\ M(k((\mu\phi(res)).\pi))(m) = \nu \end{array}\right\}\right)$$
$$= \alpha_{\tau|_{res}}\left(\left\{\phi \star \mu \in \Sigma_{\tau|_{res}} \;\middle|\; \begin{array}{l} \phi \in \overline{\phi}_{\tau|_{res}}(e),\ \mu \in \overline{\mu}(e) \\ M(k((\mu\phi(res)).\pi))(m) = \nu \end{array}\right\}\right)\ ,$$

which, by Corollary 65, is

$$\cup\{\{\pi\} \cup \delta_{F(k(\pi))}(e) \mid \pi \in e,\ k(\pi) \leq \tau(res),\ M(k(\pi))(m) = \nu\}\ .$$

<u>put_field</u>

$$\begin{aligned}
&\alpha_{\tau|_{-res}}(\mathsf{put\_field}_{\tau,\tau'}(\gamma_\tau(e_1))(\gamma_\tau(e_2))) \\
&= \alpha_{\tau|_{-res}}(\mathsf{put\_field}_{\tau,\tau'}(\{\sigma_1 \in \Sigma_\tau \mid \alpha_\tau(\sigma_1) \subseteq e_1\}) \\
&\qquad\qquad (\{\sigma_2 \in \Sigma_{\tau'} \mid \alpha_{\tau'}(\sigma_2) \subseteq e_2\})) \\
&= \alpha_{\tau|_{-res}}\left(\left\{\begin{array}{l} \phi_2|_{-res} \star \mu_2[l \mapsto \mu_2(l).\pi\, \star \\ \star\, \mu_2(l).\phi[f \mapsto \phi_2(res)]] \end{array} \;\middle|\; \begin{array}{l} \phi_1 \star \mu_1 \in \Sigma_\tau \\ \phi_2 \star \mu_2 \in \Sigma_{\tau'} \\ \alpha_\tau(\phi_1 \star \mu_1) \subseteq e_1 \\ \alpha_{\tau'}(\phi_2 \star \mu_2) \subseteq e_2 \\ (l = \phi_1(res)) \neq null \\ \mu_1 =_l \mu_2 \end{array}\right\}\right) \qquad (17)
\end{aligned}$$

which is $\varnothing$ if there is no $\pi \in e_1$ such that $k(\pi) \leq \tau(res)$, since in such a case the condition $\phi_1(res) \neq null$ cannot be satisfied. Otherwise, note that the operation put_field copies the value of $\phi_2(res)$, which is obviously reachable from ϕ_2, inside a field. Since $\alpha_{\tau'}(\phi_2 \star \mu_2) \subseteq e_2$, we conclude that an upper bound of (17) is e_2. Then $\delta_{\tau|_{-res}}(e_2)$ is also an upper bound of (17) (Corollary 66). We show that it is also a lower bound. Let $\pi_1 \in e_1$ be such that $k(\pi_1) \leq \tau(\mathtt{this})$ (possible for P1) and

$\pi_2 \in e_1$ be such that $k(\pi_2) \leq \tau(\mathit{res})$ (possible for the hypothesis on e_1). Let $o_1 = \pi_1 \star \Im(F(k(\pi_1)))$ and $o_2 = \pi_2 \star \Im(F(k(\pi_2)))$. We obtain the following lower bound of (17) by choosing special cases for ϕ_1, μ_1, ϕ_2 and μ_2:

$$\alpha_{\tau|_{-res}}\left(\left\{ \begin{array}{c} \phi_2|_{-res}\star\mu_2[l_2 \mapsto \mu_2(l_2).\pi \star \\ \star\,\mu_2(l_2).\phi[f \mapsto \phi_2(\mathit{res})]] \end{array} \middle| \begin{array}{l} \phi_1 = \Im(\tau)[\mathtt{this} \mapsto l_1, \mathit{res} \mapsto l_2] \\ \phi_2 \in \overline{\phi}_{\tau'}(e_2),\ \mu'_2 \in \overline{\mu}(e_2) \\ \phi_2 \star \mu'_2 \in \Sigma_{\tau'} \\ \mu_1 = \mu_2 = \mu'_2[l_1 \mapsto o_1, l_2 \mapsto o_2] \\ l_1, l_2 \in \mathit{Loc} \setminus \mathrm{dom}(\mu'_2),\ l_1 \neq l_2 \end{array} \right\}\right). \tag{18}$$

Since l_2 is not used in ϕ_2 nor in μ'_2, (18) becomes

$$\begin{aligned} &\alpha_{\tau|_{-res}}\left(\left\{ \phi_2|_{-res} \star \mu_2 \in \Sigma_{\tau|_{-res}} \middle| \begin{array}{l} \phi_2 \in \overline{\phi}_{\tau'}(e_2) \\ \mu_2 \in \overline{\mu}(e_2) \end{array} \right\}\right) \\ \text{(Definition 61)} &= \alpha_{\tau|_{-res}}(\{\phi \star \mu \in \Sigma_{\tau|_{-res}} \mid \phi \in \overline{\phi}_{\tau|_{-res}}(e_2),\ \mu \in \overline{\mu}(e_2)\}) \\ \text{(Lemma 63)} &= \delta_{\tau|_{-res}}(e_2)\ . \end{aligned}$$

$\underline{\cup}$
By additivity (Proposition 31), the best approximation of $\cup$ over $\wp(\Sigma_\tau)$ is $\cup$ over $\wp(\Pi)$.

B. Proofs of Propositions 46, 47, 50 and 56 in Section 5.

Proposition 46. *The abstract garbage collector ξ_τ is an lco.*

Proof. By Definition 44, the map ξ_τ is reductive and monotonic. For idempotency, we have $\xi_\tau\xi_\tau(\bot) = \bot = \xi_\tau(\bot)$. Let $s \in \mathit{Frame}^{\mathcal{ER}}_\tau \times \mathit{Memory}^{\mathcal{ER}}$. If $\mathtt{this} \in \mathsf{dom}(\tau)$ and $\phi(\mathtt{this}) = \varnothing$ then $\xi_\tau\xi_\tau(s) = \bot = \xi_\tau(\bot)$. Otherwise, we prove that $\rho_\tau\xi_\tau(s) = \rho_\tau(s)$, which entails the thesis by Definition 44. We have

$$\begin{aligned} \rho_\tau\xi_\tau(\phi\star\mu) &= \rho_\tau(\phi\star\cup\{\mu|_{\mathsf{dom}(F(k(\pi')))} \mid \pi' \in \rho_\tau(\phi\star\mu)\}) \\ &= \{\pi \in \phi(v) \mid v \in \mathsf{dom}(\tau),\ \tau(v) \in \mathcal{K}\} \cup \\ &\qquad \cup \left\{ \pi \in \mu(f) \middle| \begin{array}{l} \pi' \in \rho_\tau(\phi\star\mu),\ f \in \mathsf{dom}(F(k(\pi'))) \\ F(k(\pi'))(f) \in \mathcal{K} \end{array} \right\} \\ &= \rho_\tau(\phi\star\mu). \end{aligned}$$

□

To prove Proposition 47, we need some preliminary definitions and results.

Let $s \in \mathit{Frame}^{\mathcal{ER}}_{\tau} \times \mathit{Memory}^{\mathcal{ER}}$. We define frames and memories which use all possible creation points allowed by s.

DEFINITION 69 *Let $\phi \in \mathit{Frame}^{\mathcal{ER}}_{\tau}$, $\mu \in \mathit{Memory}^{\mathcal{ER}}$ and $l : \Pi \mapsto \mathit{Loc}$ be one-to-one. We define*

$$\overline{\phi}_{\tau} = \left\{ \phi^{\flat} \in \mathit{Frame}_{\tau} \;\middle|\; \begin{array}{l} \textit{for every } v \in \mathrm{dom}(\tau) \\ \;\textit{if } \tau(v) = \mathit{int} \textit{ then } \phi^{\flat}(v) = 0 \\ \;\textit{if } \tau(v) \in \mathcal{K} \textit{ and } \phi(v) = \varnothing \textit{ then } \phi^{\flat}(v) = \mathit{null} \\ \;\textit{if } \tau(v) \in \mathcal{K} \textit{ and } \phi(v) \neq \varnothing \textit{ then } \phi^{\flat}(v) \in l\phi(v) \end{array} \right\} ,$$

$$\overline{\mu} = \left\{ \mu^{\flat} \in \mathit{Memory} \;\middle|\; \begin{array}{l} \mathrm{dom}(\mu^{\flat}) = \mathsf{rng}(l),\ \mu^{\flat}(l(\pi)) = \pi \star \phi^{\flat}_{\pi} \\ \textit{with } \phi^{\flat}_{\pi} \in \overline{\mu}_{F(\pi)} \textit{ for every } \pi \in \Pi \end{array} \right\} .$$

Lemma 70 is needed in the proof of Lemma 71.

LEMMA 70 *Let $\phi \in \mathit{Frame}^{\mathcal{ER}}_{\tau}$, $\mu \in \mathit{Memory}^{\mathcal{ER}}$, $\phi^{\flat} \in \overline{\phi}_{\tau}$ and $\mu^{\flat} \in \overline{\mu}$. Then $\varepsilon_{\tau}(\phi^{\flat} \star \mu^{\flat}) \subseteq \phi$.*

Proof. For every $v \in \mathrm{dom}(\tau)$ we have

$$\begin{aligned} \varepsilon_{\tau}(\phi^{\flat} \star \mu^{\flat})(v) &= \begin{cases} * & \text{if } \tau(v) = \mathit{int} \\ \{(\mu^{\flat}\phi^{\flat}(v)).\pi\} & \text{if } \tau(v) \in \mathcal{K} \text{ and } \phi^{\flat}(v) \in \mathit{Loc} \\ \varnothing & \text{otherwise} \end{cases} \\ \text{(Definition 69)} &= \begin{cases} * & \text{if } \tau(v) = \mathit{int} \\ \{\mu^{\flat}(l(\pi')).\pi\} & \text{if } \tau(v) \in \mathcal{K},\ \phi^{\flat}(v) \in \mathit{Loc},\ \pi' \in \phi(v) \\ \varnothing & \text{otherwise} \end{cases} \\ &= \begin{cases} * & \text{if } \tau(v) = \mathit{int} \\ \{\pi'\} & \text{if } \tau(v) \in \mathcal{K},\ \phi^{\flat}(v) \in \mathit{Loc},\ \pi' \in \phi(v) \\ \varnothing & \text{otherwise} \end{cases} \\ &\subseteq \phi(v) \ . \end{aligned}$$

□

We prove now some properties of the frames and memories of Definition 69.

LEMMA 71 *Let $\phi \in \mathit{Frame}^{\mathcal{ER}}_{\tau}$, $\mu \in \mathit{Memory}^{\mathcal{ER}}$, $\phi^{\flat} \in \overline{\phi}_{\tau}$ and $\mu^{\flat} \in \overline{\mu}$. Then*

i) $\phi^{\flat} \star \mu^{\flat} : \tau$;

ii) $\phi^\flat \star \mu^\flat \in \Sigma_\tau$ *if and only if* `this` $\notin \mathrm{dom}(\tau)$ *or* $\phi(\texttt{this}) \neq \varnothing$*;*

iii) If $\phi^\flat \star \mu^\flat \in \Sigma_\tau$ *then* $\alpha_\tau(\phi^\flat \star \mu^\flat) \subseteq \phi \star \mu$*.*

Proof.

i) Condition 1 of Definition 14 holds since $\mathsf{rng}(\phi^\flat) \cap Loc \subseteq \mathsf{rng}(l) = \mathrm{dom}(\mu^\flat)$. Moreover, if $v \in \mathrm{dom}(\phi^\flat)$ and $\phi^\flat(v) \in Loc$ then $\phi^\flat(v) \in l\phi(v)$. Thus there exists $\pi \in \phi(v)$ with $(\mu^\flat\phi^\flat(v)).\pi = \pi$ and such that $k((\mu^\flat\phi^\flat(v)).\pi) = k(\pi) \leq \tau(v)$. Condition 2 holds since if $o \in \mathsf{rng}(\mu^\flat)$ then $o.\phi = \phi^\flat_\pi$ for some $\pi \in \Pi$. Since $\phi^\flat_\pi \in \overline{\mu}_{F(k(\pi))}$, reasoning as above we have that $\phi^\flat_\pi$ is weakly $F(k(\pi))$-correct *w.r.t.* $\mu^\flat$. Then $\phi^\flat \star \mu^\flat : \tau$.

ii) By point i, we know that $\phi^\flat \star \mu^\flat : \tau$. From Definition 16, we have $\phi^\flat \star \mu^\flat \in \Sigma_\tau$ if and only if `this` $\notin \mathrm{dom}(\tau)$ or $\phi^\flat(\texttt{this}) \neq$ *null*. By Definition 69, the latter case holds if and only if $\phi(\texttt{this}) \neq \varnothing$.

iii) By Definition 41 we have

$$\begin{aligned} \alpha_\tau(\phi^\flat \star \mu^\flat) &= \varepsilon_\tau(\phi^\flat \star \mu^\flat) \star \varepsilon_{\overline{\tau}}(\{\overline{o.\phi} \star \mu^\flat \mid o \in O_\tau(\phi^\flat \star \mu^\flat)\}) \\ \text{(Lemma 70)} &\subseteq \phi \star \varepsilon_{\overline{\tau}}(\{\overline{o.\phi} \star \mu^\flat \mid o \in O_\tau(\phi^\flat \star \mu^\flat)\}) \ . \end{aligned}$$

By Definition 69, for every $o \in O_\tau(\phi^\flat \star \mu^\flat)$ we have $o.\phi \in \overline{\mu}_{F(k(\pi))}$ and hence $\overline{o.\phi} \subseteq \phi'$ with $\phi' \in \overline{\mu}_{\overline{\tau}}$. Then we have $\varepsilon_{\overline{\tau}}(\overline{o.\phi} \star \mu^\flat) \subseteq \varepsilon_{\overline{\tau}}(\phi' \star \mu^\flat)$, which by Lemma 70 is contained in μ.

□

Lemma 72 states that, given an abstract state s, if a creation point π belongs to $\rho^i(s)$ then there is a concrete state σ from those in Definition 69 and an object in $O^i(\sigma)$ created in π, and vice versa. In other words, $\rho^i(s)$ collects all and only the creation points of the objects which can ever be reached in a concrete state approximated by s.

Lemma 72 *Let* $\phi \in Frame^{\mathcal{ER}}_\tau$ *be such that if* `this` $\in \mathrm{dom}(\tau)$ *then* $\phi(\texttt{this}) \neq \varnothing$*,* $\mu \in Memory^{\mathcal{ER}}$ *and* $i \in \mathbb{N}$*. Then* $\pi \in \rho^i_\tau(\phi \star \mu)$ *if and only if there exist* $\phi^\flat \in \overline{\phi}_\tau$ *and* $\mu^\flat \in \overline{\mu}$ *such that* $\pi = o.\pi$ *for a suitable* $o \in O^i_\tau(\phi^\flat \star \mu^\flat)$*.*

Proof. We proceed by induction on i. If $i = 0$ the result holds since $\rho^0_\tau(\phi \star \mu) = \varnothing$ and for every $\phi^\flat \in \overline{\phi}_\tau$ and $\mu \in \overline{\mu}$ we have $O^0_\tau(\phi^\flat \star \mu^\flat) = \varnothing$. Assume that it holds for a given $i \in \mathbb{N}$. We have $\pi \in \rho^{i+1}_\tau(\phi \star \mu)$ if and only if $\pi \in \phi(v)$ with $v \in \mathrm{dom}(\tau)$ (and hence $\tau(v) \in \mathcal{K}$) or

$\pi \in \rho^i_{F(k(\pi))}(\phi|_{\mathrm{dom}(F(k(\pi')))} \star \mu)$ with $v \in \mathrm{dom}(\tau)$ and $\pi' \in \phi(v)$ (and hence $\tau(v) \in \mathcal{K}$). The first case holds if and only if $o.\pi = \pi$ with $o = \mu^\flat \phi^\flat(v)$, $v \in \mathrm{dom}(\tau)$ and $\phi^\flat(v) \in \mathit{Loc}$ for suitable $\phi^\flat \in \overline{\phi}_\tau$ and $\mu^\flat \in \overline{\mu}$ (Definition 69). By inductive hypothesis, the second case holds if and only if there exist $\phi^\flat_1 \in \overline{\phi}_{F(k(\pi'))}$ and $\mu^\flat \in \overline{\mu}$ such that $\pi = o.\pi$ for a suitable $o \in O^i_{F(k(\pi'))}(\phi^\flat_1 \star \mu^\flat)$, if and only if (Definition 69) there exist $\phi^\flat \in \overline{\phi}_\tau$ and $\mu^\flat \in \overline{\mu}$ such that $\pi = o.\pi$, $v \in \mathrm{dom}(\tau)$, $\phi^\flat(v) \in \mathit{Loc}$, $o' = \mu^\flat \phi^\flat(v)$ and $o \in O^i_{F(k(o'.\pi))}(o'.\phi \star \mu^\flat)$. Together, the first or the second case hold if and only if there exist $\phi^\flat \in \overline{\phi}_\tau$ and $\mu \in \overline{\mu}$ such that $o \in O^{i+1}_\tau(\phi^\flat \star \mu^\flat)$ and $o.\pi = \pi$ (Definition 21). $\square$

Lemma 73 says that the concrete states constructed through the frames and memories of Definition 69 represent a worst-case *w.r.t.* the set of creation points of the objects reachable in every concrete state.

LEMMA 73 *Let $\phi \star \mu \in \Sigma_\tau$, $i \in \mathbb{N}$ and $\phi^\# \star \mu^\# = \alpha^{\mathcal{ER}}_\tau(\phi \star \mu)$. If $o \in O^i_\tau(\phi \star \mu)$ then there exist $\phi^\flat \in \overline{\phi^\#}_\tau$ and $\mu^\flat \in \overline{\mu^\#}$ such that $o' \in O^i_\tau(\phi^\flat \star \mu^\flat)$ and $o'.\pi = o.\pi$.*

Proof. We proceed by induction on i. We have $O^0_\tau(\phi \star \mu) = \varnothing$ and the result holds for $i = 0$. Assume that it holds for a given $i \in \mathbb{N}$. Let $o \in O^{i+1}_\tau(\phi \star \mu)$. We have $o = \mu\phi(v)$ with $v \in \mathrm{dom}(\tau)$ and $\phi(v) \in \mathit{Loc}$ or $o \in O^i_{F(k(o'.\pi))}(o'.\phi \star \mu)$ with $v \in \mathrm{dom}(\tau)$, $\phi(v) \in \mathit{Loc}$ and $o' = \mu\phi(v)$. In the first case, we have $o.\pi \in \phi^\#(v)$ and there exist $\phi^\flat \in \overline{\phi^\#}_\tau$ and $\mu^\flat \in \overline{\mu^\#}$ such that $\mu^\flat\phi^\flat(v).\pi = \pi$ and the thesis follows by letting $o' = \mu^\flat\phi^\flat(v)$. In the second case, by inductive hypothesis we know that there exist $\phi^\flat_1 \in \overline{\phi^\#}_{F(k(o'.\pi))}$ and $\mu^\flat \in \overline{\mu^\#}$ such that $o'' \in O^i_{F(k(o'.\pi))}(\phi^\flat_1 \star \mu^\flat)$, $o''.\pi = o.\pi$, $v \in \mathrm{dom}(\tau)$, $\phi(v) \in \mathit{Loc}$ and $o' = \mu\phi(v)$ if and only if (Definitions 69 and 21) there exist $\phi^\flat \in \overline{\phi^\#}_\tau$ and $\mu^\flat \in \overline{\mu^\#}$ such that $o'' \in O^{i+1}_\tau(\phi^\flat \star \mu^\flat)$ and $o''.\pi = o.\pi$. $\square$

Lemma 74 gives an explicit definition of the abstraction of the set of states constructed from the frames and memories of Definition 69.

LEMMA 74 *Let $\phi \in \mathit{Frame}^{\mathcal{ER}}_\tau$ and $\mu \in \mathit{Memory}^{\mathcal{ER}}$. Then*

$$\alpha^{\mathcal{ER}}_\tau(\{\phi^\flat \star \mu^\flat \in \Sigma_\tau \mid \phi^\flat \in \overline{\phi}_\tau \text{ and } \mu^\flat \in \overline{\mu}\}) = \xi_\tau(\phi \star \mu) \ .$$

Proof. Let $A_\tau = \alpha^{\mathcal{ER}}_\tau(\{\phi^\flat \star \mu^\flat \in \Sigma_\tau \mid \phi^\flat \in \overline{\phi}_\tau \text{ and } \mu^\flat \in \overline{\mu}\})$. If `this` $\in \mathrm{dom}(\tau)$ and $\phi($`this`$) = \varnothing$, then $A_\tau = \bot$ because of Lemma 71.ii.

Moreover, $\xi_\tau(\phi\star\mu)=\perp$ (Definition 44). Otherwise, by Definition 69 we have

$$\begin{aligned}A_\tau &= \epsilon_\tau\left(\left\{\phi^\flat\star\mu^\flat \;\middle|\; \begin{array}{l}\phi^\flat\in\overline{\phi}_\tau\\ \mu^\flat\in\overline{\mu}\end{array}\right\}\right)\star\epsilon_{\overline{\tau}}\left(\left\{\overline{o.\phi}\star\mu^\flat \;\middle|\; \begin{array}{l}\phi^\flat\in\overline{\phi}_\tau,\ \mu^\flat\in\overline{\mu},\\ o\in O_\tau(\phi^\flat\star\mu^\flat)\end{array}\right\}\right)\\ &= \phi\star\epsilon_{\overline{\tau}}(\{\overline{\phi'}\star\mu^\flat\mid\phi'\in\overline{\mu}_{F(k(o.\pi))},\ \phi^\flat\in\overline{\phi}_\tau,\ \mu^\flat\in\overline{\mu},\ o\in O_\tau(\phi^\flat\star\mu^\flat)\})\\ &= \phi\star\epsilon_{\overline{\tau}}(\{\overline{\phi'}\star\mu'\mid\phi'\in\overline{\mu}_{F(k(o.\pi))},\ \phi^\flat\in\overline{\phi}_\tau,\ \mu',\mu^\flat\in\overline{\mu},\ o\in O_\tau(\phi^\flat\star\mu^\flat)\})\end{aligned} \tag{19}$$

since $\epsilon_{\overline{\tau}}$ does not depend on the frames of the objects in memory (Definition 37). By Lemma 72, (19) is equal to

$$\begin{aligned}&\phi\star\epsilon_{\overline{\tau}}(\{\overline{\phi'}\star\mu'\mid\phi'\in\overline{\mu}_{F(k(\pi))},\ \mu'\in\overline{\mu},\ \pi\in\rho_\tau(\phi\star\mu)\})\rangle\\ &=\phi\star\cup\{\mu|_{\mathrm{dom}(F(k(\pi)))}\mid\pi\in\rho_\tau(\phi\star\mu)\}\cup\Im(\overline{\tau})\\ &=\xi_\tau(\phi\star\mu)\ .\end{aligned}$$

□

We now prove Proposition 47. To do this, we will use the set of states constructed from the frames and memories in Definition 69 to show that $\alpha^{\mathcal{ER}}$ is onto.

Proposition 47. *Let ξ_τ be the abstract garbage collector of Definition 44. Then $\mathsf{fp}(\xi_\tau)=\mathsf{rng}(\alpha^{\mathcal{ER}}_\tau)$.*

Proof. Let $X\subseteq\Sigma_\tau$. By Proposition 46, Lemmas 73 and 74 and Definition 41, we have

$$\begin{aligned}\alpha^{\mathcal{ER}}_\tau(X)&=\cup\{\alpha^{\mathcal{ER}}_\tau(\sigma)\mid\sigma\in X\}\\ &\subseteq\cup\,\alpha^{\mathcal{ER}}_\tau\left(\left\{\phi^\flat\star\mu^\flat\in\Sigma_\tau\;\middle|\;\begin{array}{l}\phi^\flat\in\overline{\alpha^{\mathcal{ER}}_\tau(\sigma).\phi_\tau}\\ \mu^\flat\in\overline{\alpha^{\mathcal{ER}}_\tau(\sigma).\mu}\\ \sigma\in X\end{array}\right\}\right)\\ &=\cup\{\xi_\tau\alpha^{\mathcal{ER}}_\tau(\sigma)\mid\sigma\in X\}\subseteq\xi_\tau\alpha^{\mathcal{ER}}_\tau(X)\ .\end{aligned}$$

The converse inclusion holds since ξ_τ is reductive (Proposition 46) and, hence $\alpha^{\mathcal{ER}}_\tau(X)\in\mathsf{fp}(\xi_\tau)$. Conversely, let $s\in\mathsf{fp}(\xi_\tau)$ and $X=\{\phi^\flat\star\mu^\flat\in\Sigma_\tau\mid\phi^\flat\in\overline{\phi}_\tau,\ \mu^\flat\in\overline{\mu}\}$. By Lemma 74 and since $s\in\mathsf{fp}(\xi_\tau)$, we have $\alpha^{\mathcal{ER}}_\tau(X)=\xi_\tau(s)=s$.

The proof of Proposition 50 requires some preliminary results.

Corollary 75 states that if we know that the approximation of a set of concrete states S is some $\phi\star\mu$, then we can conclude that a better approximation of S is $\xi(\phi\star\mu)$. In other words, garbage is not used in the approximation.

Corollary 75 *Let $S \subseteq \Sigma_\tau$, $\phi \in Frame_\tau^{\mathcal{ER}}$ and $\mu \in Memory^{\mathcal{ER}}$. Then $\alpha_\tau(S) \subseteq \xi_\tau(\phi \star \mu)$ if and only if $\alpha_\tau(S) \subseteq \phi \star \mu$.*

Proof. Assume that $\alpha_\tau(S) \subseteq \xi_\tau(\phi \star \mu)$. By reductivity (Proposition 46) we have $\alpha_\tau(S) \subseteq \phi \star \mu$. Conversely, assume that $\alpha_\tau(S) \subseteq \phi \star \mu$. By Proposition 47 and monotonicity (Proposition 46) we have $\alpha_\tau(S) = \xi_\tau \alpha_\tau(S) \subseteq \xi_\tau(\phi \star \mu)$. □

The following lemma will be used in the proof of Proposition 50. It states that the approximation of a variable depends from the concrete value of that variable only, and that the approximation of a memory is the same if the locations in the frame do not change (although they may be bound to different variables).

Lemma 76 *Let $\phi' \star \mu \in \Sigma_{\tau'}$ and $\phi'' \star \mu \in \Sigma_{\tau''}$. Then*

i) if $\phi'(v) = \phi''(v)$ for each $v \in \mathrm{dom}(\tau') \cap \mathrm{dom}(\tau'')$, then we have $(\alpha_{\tau'}(\phi' \star \mu)).\phi(v) = (\alpha_{\tau''}(\phi'' \star \mu)).\phi(v)$;

ii) if $\mathsf{rng}(\phi') \cap Loc = \mathsf{rng}(\phi'') \cap Loc$, then we have $(\alpha_{\tau'}(\phi' \star \mu)).\mu = (\alpha_{\tau''}(\phi'' \star \mu)).\mu$.

Proof. From Definition 41. □

Lemma 77 says that if we consider all the concrete states approximated by some $\phi^\# \star \mu^\#$ and we restrict their frames, the resulting set of states is approximated by $\xi(\phi^\# \star \mu^\#)$. In other words, the operation ξ garbage collects all objects that, because of the restriction, are no longer reachable.

Lemma 77 *Let $vs \subseteq \mathrm{dom}(\tau)$ and $\phi^\# \star \mu^\# \in \mathcal{ER}_\tau$. Then*

$$\alpha_{\tau|_{-vs}}\left(\left\{\phi|_{-vs} \star \mu \,\middle|\, \begin{array}{l} \phi \star \mu \in \Sigma_\tau \\ \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\# \end{array}\right\}\right) = \xi_{\tau|_{-vs}}(\phi^\#|_{-vs} \star \mu^\#) \ .$$

Proof. We have

$$\begin{aligned} &\alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\#\}) \\ &= \alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \star \mu \in \Sigma_{\tau|_{-vs}} \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\#\}) \ , \end{aligned} \tag{20}$$

since if $\phi \star \mu \in \Sigma_\tau$ then $\phi|_{-vs} \star \mu \in \Sigma_{\tau|_{-vs}}$. We have that, if $\alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\#$, then $\alpha_{\tau|_{-vs}}(\phi|_{-vs} \star \mu) \subseteq \phi^\#|_{-vs} \star \mu^\#$. Hence (20) is contained in $\phi^\#|_{-vs} \star \mu^\#$. By Corollary 75, the set (20) is also contained in the set $\xi_{\tau|_{-vs}}(\phi^{\#}|_{-vs} \star \mu^{\#})$. But also the converse inclusion holds, since in (20) we can restrict the choice of $\phi \star \mu \in \Sigma_\tau$, so that (20) contains

$$\alpha_{\tau|_{-vs}}\left(\left\{\phi|_{-vs} \star \mu \in \Sigma_{\tau|_{-vs}} \;\middle|\; \begin{array}{l} \phi \star \mu \in \Sigma_\tau,\ \alpha_\tau(\phi \star \mu) \subseteq \phi^{\#} \star \mu^{\#} \\ \phi \in \overline{\phi^{\#}}_\tau,\ \mu \in \overline{\mu^{\#}} \end{array}\right\}\right) . \tag{21}$$

By points ii and iii of Lemma 62, (21) is equal to

$$\begin{aligned} &\alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \star \mu \in \Sigma_{\tau|_{-vs}} \mid \phi \in \overline{\phi^{\#}}_\tau,\ \mu \in \overline{\mu^{\#}}\}) \\ (\text{Definition 69}) = {}&\alpha_{\tau|_{-vs}}\left(\left\{\phi \star \mu \in \Sigma_{\tau|_{-vs}} \;\middle|\; \begin{array}{l} \phi \in \overline{(\phi^{\#}|_{-vs})}_{\tau|_{-vs}} \\ \mu \in \overline{\mu^{\#}} \end{array}\right\}\right) \\ (\text{Lemma 74}) = {}&\xi_{\tau|_{-vs}}(\phi^{\#}|_{-vs} \star \mu^{\#}) \ . \end{aligned}$$

□

We are now ready to prove the correctness and optimality of the abstract operations in Figure 10.

Proposition 50. *The operations in Figure 10 are the optimal counterparts induced by $\alpha^{\mathcal{ER}}$ of the operations in Figure 8 and of $\cup$.*

Proof. The strictness of the abstract operations (except $\cup$) follows by reasoning as for the proof of strictness in Proposition 32. Note that $\gamma_\tau(\bot) = \varnothing$ for all $\tau \in \mathit{TypEnv}$ since, by Definition 41,

$$\gamma_\tau(\bot) = \{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq \bot\} = \{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) = \bot\} = \varnothing.$$

Hence return is also strict on both arguments.

We will use the corresponding versions of the properties P2 and P3 already used in the proof of Proposition 32. They are

P2 If $\phi \star \mu \in \mathcal{ER}_\tau$ then there exists $\sigma \in \Sigma_\tau$ such that $\alpha_\tau(\sigma) \subseteq \phi \star \mu$.

P3 $\alpha_\tau \gamma_\tau$ is the identity map.

P2 holds since $\phi(\mathtt{this}) \neq \varnothing$ so that there exists $\pi \in \phi(\mathtt{this})$ and hence, letting $\sigma = [\mathtt{this} \mapsto l] \star [l \mapsto \pi \star \Im(F(k(\pi)))]$ for some $l \in \mathit{Loc}$, we have $\sigma \in \Sigma_\tau$. Moreover, $\alpha_\tau(\sigma) = \phi^\bot[\mathtt{this} \mapsto \{\pi\}] \star \mu^\bot \subseteq \phi \star \mu$, where $\phi^\bot$ and $\mu^\bot$ are the least elements of $\mathit{Frame}_\tau^{\mathcal{ER}}$ and $\mathit{Memory}^{\mathcal{ER}}$, respectively. By Proposition 49, α_τ is a Galois insertion and hence, P3 holds.

Most cases of the proof are similar to the corresponding cases in the proof of Proposition 32, provided we use Lemma 76 instead of Lemma 67, Lemma 77 instead of Lemma 68, Definition 44 instead of

Definition 25, and we modify the syntax of the abstract elements. As an example, consider

<u>get_int, get_null, get_var</u>

$$\begin{aligned}
&\alpha_{\tau[res\mapsto int]}(\mathsf{get\_int}^i_\tau(\gamma_\tau(\phi^\# \star \mu^\#)))\\
&= \alpha_{\tau[res\mapsto int]}(\{\phi'[res \mapsto i] \star \mu' \mid \phi' \star \mu' \in \gamma_\tau(\phi^\# \star \mu^\#)\})\\
(*) &= \alpha_\tau(\{\phi' \star \mu' \mid \phi' \star \mu' \in \gamma_\tau(\phi^\# \star \mu^\#)\}).\phi[res \mapsto *] \star\\
&\quad \star \alpha_\tau(\{\phi' \star \mu' \mid \phi' \star \mu' \in \gamma_\tau(\phi^\# \star \mu^\#)\}).\mu\\
(\text{P3}) &= \phi^\#[res \mapsto *] \star \mu^\# \ .
\end{aligned}$$

where point $*$ follows by Lemma 76 since $res \notin \mathrm{dom}(\tau)$ and $\tau[res \mapsto int](res) = int$. The proof is similar for get_null and get_var.

Therefore, we only show the cases which differ significantly from the corresponding case in Proposition 32.

<u>is_null</u>
Let $A = \alpha_{\tau[res\mapsto int]}(\mathsf{is\_null}_\tau(\gamma_\tau(\phi^\# \star \mu^\#)))$. We have

$$\begin{aligned}
A &= \alpha_{\tau[res\mapsto int]}(\mathsf{is\_null}_\tau(\{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq \phi^\# \star \mu^\#\}))\\
&= \alpha_{\tau[res\mapsto int]}(\{\phi[res \mapsto 1] \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\#\}) \ .
\end{aligned}$$

By Lemma 76.i we have

$$\begin{aligned}
A.\phi &= \phi^\#[res \mapsto *]\\
(\text{Definition 44}) &= \xi_{\tau[res\mapsto int]}(\phi^\#[res \mapsto *] \star \mu^\#) \ .
\end{aligned}$$

Moreover, by Lemma 76.ii we have

$$\begin{aligned}
A.\mu &= \alpha_{\tau|_{-res}}\left(\left\{ \phi|_{-res} \star \mu \,\middle|\, \begin{array}{l} \phi \star \mu \in \Sigma_\tau \\ \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\# \end{array} \right\}\right).\mu\\
(\text{Lemma 77}) &= \xi_{\tau|_{-res}}(\phi^\#|_{-res} \star \mu^\#).\mu\\
(\text{Definition 44}) &= \xi_{\tau[res\mapsto int]}(\phi^\#[res \mapsto *] \star \mu^\#).\mu \ .
\end{aligned}$$

<u>put_var</u>
Let $A = \alpha_{\tau|_{-res}}(\mathsf{put\_var}_\tau(\gamma_\tau(\phi^\# \star \mu^\#)))$. We have

$$\begin{aligned}
A &= \alpha_{\tau|_{-res}}(\mathsf{put\_var}_\tau(\{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq \phi^\# \star \mu^\#\}))\\
&= \alpha_{\tau|_{-res}}\left(\left\{ \phi[v \mapsto \phi(res)]|_{-res} \star \mu \,\middle|\, \begin{array}{l} \phi \star \mu \in \Sigma_\tau \\ \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\# \end{array} \right\}\right) \ .
\end{aligned}$$

By Lemma 76.i we have

$$\begin{aligned} A.\phi &= \phi^{\#}[v \mapsto \phi^{\#}(\mathit{res})]|_{-\mathit{res}} \\ \text{(Definition 44)} &= \xi_{\tau|_{-\mathit{res}}}(\phi^{\#}[v \mapsto \phi^{\#}(\mathit{res})]|_{-\mathit{res}} \star \mu^{\#}).\phi\ . \end{aligned}$$

Moreover, since $\mathsf{rng}(\phi[v \mapsto \phi(\mathit{res})]|_{-\mathit{res}}) = \mathsf{rng}(\phi|_{-v})$, by Lemma 76.ii we have

$$\begin{aligned} A.\mu &= \alpha_{\tau|_{-v}}\left(\left\{ \phi|_{-v} \star \mu \;\middle|\; \begin{array}{l} \phi \star \mu \in \Sigma_\tau \\ \alpha_\tau(\phi \star \mu) \subseteq \phi^{\#} \star \mu^{\#} \end{array} \right\}\right).\mu \\ \text{(Lemma 77)} &= \xi_{\tau|_{-v}}(\phi^{\#}|_{-v} \star \mu^{\#}).\mu \\ \text{(Definition 44)} &= \xi_{\tau|_{-\mathit{res}}}(\phi^{\#}[v \mapsto \phi^{\#}(\mathit{res})]|_{-\mathit{res}} \star \mu^{\#}).\mu\ . \end{aligned}$$

call

Let $p = P(\nu)|_{-\mathtt{out}}$. Recall that $\mathrm{dom}(p) = \{\iota_1, \ldots, \iota_n, \mathtt{this}\}$. Let $\tau\prime = \tau[v_1 \mapsto \iota_1, \ldots, v_n \mapsto \iota_n, \mathit{res} \mapsto \mathtt{this}]$ and $\phi^{\#}_{\prime} = \phi^{\#}[v_1 \mapsto \iota_1, \ldots, v_n \mapsto \iota_n, \mathit{res} \mapsto \mathtt{this}]$. We have

$$\begin{aligned} & \alpha_p(\mathsf{call}^{\nu, v_1, \ldots, v_n}_\tau(\gamma_\tau(\phi^{\#} \star \mu^{\#}))) \\ &= \alpha_p(\mathsf{call}^{\nu, v_1, \ldots, v_n}_\tau(\{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq \phi^{\#} \star \mu^{\#}\})) \\ &= \alpha_p\left(\left\{ \begin{bmatrix} \iota_1 \mapsto \phi(v_1), \\ \vdots \\ \iota_n \mapsto \phi(v_n), \\ \mathtt{this} \mapsto \phi(\mathit{res}) \end{bmatrix} \star \mu \;\middle|\; \begin{array}{l} \phi \star \mu \in \Sigma_\tau \text{ and} \\ \alpha_\tau(\phi \star \mu) \subseteq \phi^{\#} \star \mu^{\#} \end{array} \right\}\right) \\ \text{(Lemma 76)} &= \alpha_p(\{\phi|_p \star \mu \mid \phi \star \mu \in \Sigma_{\tau\prime} \text{ and } \alpha_{\tau\prime}(\phi \star \mu) \subseteq \phi^{\#}_{\prime} \star \mu^{\#}\}) \\ \text{(Lemma 77)} &= \xi_p(\phi^{\#}_{\prime}|_p \star \mu^{\#}) \\ &= \xi_p\left(\begin{bmatrix} \iota_1 \mapsto \phi^{\#}(v_1), \\ \vdots \\ \iota_n \mapsto \phi^{\#}(v_n), \\ \mathtt{this} \mapsto \phi^{\#}(\mathit{res}) \end{bmatrix} \star \mu^{\#} \right)\ . \end{aligned}$$

new

Let $\kappa = k(\pi)$ and $A = \alpha_{\tau[\mathit{res} \mapsto \kappa]}(\mathsf{new}^{\pi}_\tau(\gamma_\tau(\phi^{\#} \star \mu^{\#})))$. Since $\mathit{res} \notin \mathrm{dom}(\tau)$ we have

$$\begin{aligned} A &= \alpha_{\tau[\mathit{res} \mapsto \kappa]}(\mathsf{new}^{\pi}_\tau(\{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq \phi^{\#} \star \mu^{\#}\})) \\ &= \alpha_{\tau[\mathit{res} \mapsto \kappa]}\left(\left\{ \begin{array}{c} \phi[\mathit{res} \mapsto l] \star \\ \star\, \mu[l \mapsto \pi \star \Im(F(\kappa))] \end{array} \;\middle|\; \begin{array}{l} \phi \star \mu \in \Sigma_\tau \\ \alpha_\tau(\phi \star \mu) \subseteq \phi^{\#} \star \mu^{\#} \\ l \in \mathit{Loc} \setminus \mathrm{dom}(\mu) \end{array} \right\}\right)\ . \end{aligned}$$

By Lemma 76.i we have

$$\begin{aligned} A.\phi &= \alpha_\tau(\{\phi\star\mu \mid \phi\star\mu\in\Sigma_\tau \text{ and } \alpha_\tau(\phi\star\mu)\subseteq\phi^{\#}\star\mu^{\#}\}).\phi[\mathit{res}\mapsto\{\pi\}]\\ &= \alpha_\tau\gamma_\tau(\phi^{\#}\star\mu^{\#}).\phi[\mathit{res}\mapsto\{\pi\}]\\ \text{(P3)} &= \phi^{\#}[\mathit{res}\mapsto\{\pi\}]\ . \end{aligned}$$

The newly created object $o=\pi\star\Im(F(\kappa))$ has its fields bound to *null*: $o.\phi(f)=\Im(F(\kappa))(f)\in\{0,\mathit{null}\}$ for every $f\in\mathrm{dom}(o.\phi)$. Hence it does not contribute to the memory component $A.\mu$ and by Lemma 76.ii we have

$$\begin{aligned} A.\mu &= \alpha_\tau(\{\phi\star\mu \mid \phi\star\mu\in\Sigma_\tau \text{ and } \alpha_\tau(\phi\star\mu)\subseteq\phi^{\#}\star\mu^{\#}\}).\mu\\ &= \alpha_\tau\gamma_\tau(\phi^{\#}\star\mu^{\#}).\mu\\ \text{(P3)} &= \mu^{\#}\ . \end{aligned}$$

<u>return</u>
Let $\tau'=\tau[\mathit{res}\mapsto P(\nu)(\texttt{out})]$, $\tau''=P(\nu)|_{\texttt{out}}$ and $L=\mathsf{rng}(\phi_1|_{-\mathit{res}})\cap \mathit{Loc}$.

$$\begin{aligned} &\alpha_{\tau'}(\mathsf{return}^\nu_\tau(\gamma_\tau(\phi_1^{\#}\star\mu_1^{\#}))(\gamma_{\tau''}(\phi_2^{\#}\star\mu_2^{\#})))\\ &= \alpha_{\tau'}(\mathsf{return}^\nu_\tau(\{\sigma_1\in\Sigma_\tau \mid \alpha_\tau(\sigma_1)\subseteq\phi_1^{\#}\star\mu_1^{\#}\})\\ &\qquad(\{\sigma_2\in\Sigma_{\tau''}\mid \alpha_{\tau''}(\sigma_2)\subseteq\phi_2^{\#}\star\mu_2^{\#}\}))\\ &= \alpha_{\tau'}\left(\left\{ \phi_1|_{-\mathit{res}}[\mathit{res}\mapsto\phi_2(\texttt{out})]\star\mu_2 \;\middle|\; \underbrace{\begin{matrix}\phi_1\star\mu_1\in\Sigma_\tau\\ \phi_2\star\mu_2\in\Sigma_{\tau''}\\ \alpha_\tau(\phi_1\star\mu_1)\subseteq\phi_1^{\#}\star\mu_1^{\#}\\ \alpha_{\tau''}(\phi_2\star\mu_2)\subseteq\phi_2^{\#}\star\mu_2^{\#}\\ \mu_1=_L\mu_2\end{matrix}}_{\mathit{Cond}} \right\}\right)\\ (*) &= \underbrace{\alpha_{\tau|_{-\mathit{res}}}(\{\phi_1|_{-\mathit{res}}\star\mu_2\mid \mathit{Cond}\})}_{A}\cup\underbrace{\alpha_{\tau''}(\{\phi_2\star\mu_2\mid\mathit{Cond}\})[\texttt{out}\mapsto\mathit{res}]}_{B} \end{aligned}$$

where point $*$ follows by Definition 41. Since $\alpha_{\tau''}(\phi_2\star\mu_2)\subseteq\phi_2^{\#}\star\mu_2^{\#}$, we have $B\subseteq\phi_2^{\#}[\texttt{out}\mapsto\mathit{res}]\star\mu_2^{\#}$. But the converse inclusion holds also, since by Lemma 71.iii we have

$$B\supseteq\alpha_{\tau''}\left(\left\{\phi_2\star\mu_2 \;\middle|\; \begin{matrix}\phi_1\in\overline{\phi_1^{\#}}_\tau,\ \mu_1\in\overline{\mu_1^{\#}}\\ \phi_2\in\overline{\phi_2^{\#}}_{\tau''},\ \mu_2\in\overline{\mu_2^{\#}}\end{matrix}\right\}\right)[\texttt{out}\mapsto\mathit{res}]$$

which by Lemma 74 is equal to $\phi_2^\#[\mathtt{out} \mapsto \mathit{res}] \star \mu_2^\#$. Note that the condition $\mu_1 =_L \mu_2$ is satisfied by Definition 69. Since $\mathrm{dom}(\tau'') = \{\mathtt{out}\}$, we conclude that $B = [\mathit{res} \mapsto \phi_2^\#(\mathtt{out})] \star \mu_2^\#$.

With regard to A, we have

$$\begin{aligned} A &\supseteq \alpha_{\tau|_{-res}} \left\{ \phi_1|_{-res} \star \mu_2 \;\middle|\; \begin{array}{l} \mathit{Cond},\ \phi_1 \in \overline{\phi_1^\#}_\tau,\ \underline{\mu_1} \in \overline{\mu_1^\#} \\ \phi_2 = \Im(\tau''),\ \mu_2 \in \overline{\mu^\top} \end{array} \right\} \\ (\text{Lemma 71}) &= \alpha_{\tau|_{-res}} \{ \phi_1|_{-res} \star \mu_2 \mid \phi_1 \in \overline{\phi_1^\#}_\tau,\ \mu_2 \in \overline{\mu^\top} \} \\ (\text{Lemma 74}) &= \xi_{\tau|_{-res}} (\phi_1^\#|_{-res} \star \mu^\top) \ . \end{aligned} \tag{22}$$

Moreover, for every $v \in \mathrm{dom}(\tau|_{-res})$ such that $\tau(v) \in \mathcal{K}$, we have

$$\begin{aligned} A.\phi(v) &= \{ o.\pi \mid \phi_1|_{-res}(v) \in \mathit{Loc},\ o = \mu_2 \phi_1|_{-res}(v),\ \mathit{Cond} \} \\ (\text{since } \mu_1 =_L \mu_2) &= \{ o.\pi \mid \phi_1|_{-res}(v) \in \mathit{Loc},\ o = \mu_1 \phi_1|_{-res}(v),\ \mathit{Cond} \} \\ &\subseteq \left\{ (\mu_1 \phi_1(v)).\pi \;\middle|\; \begin{array}{l} \phi_1(v) \in \mathit{Loc},\ \phi_1 \star \mu_1 \in \Sigma_\tau \\ \alpha_\tau(\phi_1 \star \mu_1) \subseteq \phi_1^\# \star \mu_1^\# \end{array} \right\} \\ &= (\alpha_\tau \gamma_\tau (\phi_1^\# \star \mu_1^\#)).\phi(v) \\ (\text{P1}) &= \phi_1^\#(v) \ . \end{aligned}$$

We conclude that $A.\phi \subseteq \phi_1^\#|_{-res}$. Moreover, we have $A.\mu \subseteq \mu^\top$. Hence $A \subseteq \phi_1^\#|_{-res} \star \mu^\top$ and, by Corollary 75, $A \subseteq \xi_{\tau|_{-res}}(\phi_1^\#|_{-res} \star \mu^\top)$. Together with (22), this proves that $A = \xi_{\tau|_{-res}}(\phi_1^\#|_{-res} \star \mu^\top)$.

<u>get_field</u>

Let $\tau' = \tau[\mathit{res} \mapsto (F\tau(\mathit{res}))(f)]$ and $A = \alpha_{\tau'}(\mathsf{get\_field}_\tau^f(\gamma_\tau(\phi^\# \star \mu^\#)))$. We have

$$\begin{aligned} A &= \alpha_{\tau'}(\mathsf{get\_field}_\tau^f(\{ \phi \star \mu \in \Sigma_\tau \mid \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\# \})) \\ &= \alpha_{\tau'} \left(\left\{ \phi[\mathit{res} \mapsto (\mu\phi(\mathit{res})).\phi(f)] \star \mu \;\middle|\; \begin{array}{l} \phi \star \mu \in \Sigma_\tau \\ \phi(\mathit{res}) \neq \mathit{null} \\ \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\# \end{array} \right\} \right) \end{aligned}$$

which is $\bot$ when $\phi^\#(\mathit{res}) = \varnothing$, since in such a case the condition $\phi(\mathit{res}) \neq \mathit{null}$ cannot be satisfied. Assume then that we have $\phi^\#(\mathit{res}) \neq$

$\varnothing$ and let $f' = (\mu\phi(res)).\phi(f)$. We conclude that

$$\begin{aligned}
A &\supseteq \alpha_{\tau'}\left(\left\{\phi[res \mapsto f'] \star \mu \;\middle|\; \begin{array}{l} \phi \star \mu \in \Sigma_\tau \\ \phi(res) \neq null \\ \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\# \\ \phi \in \overline{\phi^\#}_\tau,\ \mu \in \overline{\mu^\#} \end{array}\right\}\right) \\
\text{(Definition 69)} &= \alpha_{\tau'}\left(\left\{\phi[res \mapsto f'] \star \mu \;\middle|\; \begin{array}{l} \phi \star \mu \in \Sigma_\tau \\ \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\# \\ \phi \in \overline{\phi^\#}_\tau,\ \mu \in \overline{\mu^\#} \end{array}\right\}\right) \\
\text{(Lemma 71)} &= \alpha_{\tau'}(\{\phi[res \mapsto f'] \star \mu \mid \phi \in \overline{\phi^\#}_\tau,\ \mu \in \overline{\mu^\#}\}) \\
\text{(Definition 69)} &= \alpha_{\tau'}(\{\phi \star \mu \mid \phi \in \overline{\phi^\#[res \mapsto \mu(f)]}_{\tau'},\ \mu \in \overline{\mu^\#}\}) \\
\text{(Lemma 74)} &= \phi^\#[res \mapsto \mu(f)] \star \mu^\# \ .
\end{aligned}$$

We prove that the converse inclusion also holds. Let $x = (\mu\phi(res)).\phi(f)$. If $x \in Loc$, the object $\mu(x)$ is reachable by construction from $\phi(res)$. Hence we have

$$\begin{aligned}
A.\mu &\subseteq \alpha_\tau(\{\phi \star \mu \mid \phi \star \mu \in \Sigma_\tau,\ \phi(res) \neq null,\ \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\#\}).\mu \\
&\subseteq \alpha_\tau(\{\phi \star \mu \mid \phi \star \mu \in \Sigma_\tau,\ \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\#\}).\mu \\
&= \alpha_\tau\gamma_\tau(\phi^\# \star \mu^\#).\mu \\
\text{(P3)} &= \mu^\# \ .
\end{aligned}$$

If $\phi(res) \neq null$ then $o = \mu\phi(res) \in O_\tau(\phi \star \mu)$ and $\varepsilon_{\overline{\tau}}(\overline{o.\phi} \star \mu) \subseteq \mu^\#$ (Definition 41). Hence, if $(\mu\phi(res)).\phi(f) \neq null$ then we have that $((\mu\phi(res)).\phi(f)).\pi \in \mu^\#(f)$. By Lemma 76 we conclude that

$$A.\phi \subseteq \phi^\#[res \mapsto \mu^\#(f)] \ .$$

<u>lookup</u>

Let $A = \alpha_\tau(\mathsf{lookup}_\tau^{m,\nu}(\gamma_\tau(\phi^\# \star \mu^\#)))$. We have

$$\begin{aligned}
A &= \alpha_\tau(\mathsf{lookup}_\tau^{m,\nu}(\{\phi \star \mu \in \Sigma_\tau \mid \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\#\})) \\
&= \alpha_\tau\left(\left\{\phi \star \mu \in \Sigma_\tau \;\middle|\; \begin{array}{l} \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\# \\ \phi(res) \neq null,\ M((\mu\phi(res)).\pi)(m) = \nu \end{array}\right\}\right) \ .
\end{aligned}$$

We have $A = \bot$ if there is no $\pi \in \phi^\#(res)$ such that $M(\pi)(m) = \nu$, because in such a case the condition $M((\mu\phi(res)).\pi)(m) = \nu$ cannot be satisfied. Otherwise we have

$$\begin{aligned}
A &\supseteq \alpha_\tau\left(\left\{\phi\star\mu\in\Sigma_\tau \;\middle|\; \begin{array}{l}\alpha_\tau(\phi\star\mu)\subseteq\phi^\#\star\mu^\#\\ \phi(\mathit{res})\neq \mathit{null}\\ M(k(\underline{(\mu\phi(\mathit{res}))}.\pi))(m)=\nu\\ \phi\in\overline{\phi^\#}_\tau,\ \mu\in\overline{\mu^\#}\end{array}\right\}\right)\\
\text{(Definition 44)} &= \alpha_\tau\left(\left\{\phi\star\mu\in\Sigma_\tau \;\middle|\; \begin{array}{l}\alpha_\tau(\phi\star\mu)\subseteq\phi^\#\star\mu^\#\\ M(k(\underline{(\mu\phi(\mathit{res}))}.\pi))(m)=\nu\\ \phi\in\overline{\phi^\#}_\tau,\ \mu\in\overline{\mu^\#}\end{array}\right\}\right)\\
\text{(Lemma 71.iii)} &= \alpha_\tau\left(\left\{\phi\star\mu\in\Sigma_\tau \;\middle|\; \begin{array}{l}M(k(\underline{(\mu\phi(\mathit{res}))}.\pi))(m)=\nu\\ \phi\in\overline{\phi^\#}_\tau,\ \mu\in\overline{\mu^\#}\end{array}\right\}\right)\\
\text{(Definition 69)} &= \alpha_\tau\left(\left\{\phi\star\mu\in\Sigma_\tau \;\middle|\; \begin{array}{l}\underline{(\mu\phi(\mathit{res}))}.\pi\in S,\\ \phi\in\overline{\phi^\#}_\tau,\ \mu\in\overline{\mu^\#}\end{array}\right\}\right)
\end{aligned}$$

where $S=\{\pi\in\phi^\#(\mathit{res})\mid M(k(\pi))(m)=\nu\}$. By Definition 69 we have

$$\begin{aligned}
A &\supseteq \alpha_\tau(\{\phi\star\mu\in\Sigma_\tau\mid \phi\in\overline{\phi^\#[\mathit{res}\mapsto S]}_\tau,\ \mu\in\overline{\mu^\#}\})\\
\text{(Lemma 74)} &= \xi_\tau(\phi^\#[\mathit{res}\mapsto S]\star\mu^\#)\ .
\end{aligned}$$

We prove that also the converse inclusion holds. Note that if $\alpha_\tau(\phi\star\mu)\subseteq\phi^\#[\mathit{res}\mapsto S]\star\mu^\#$ and $\phi(\mathit{res})\neq\mathit{null}$ then $M(k(\underline{(\mu\phi(\mathit{res}))}.\pi))(m)=\nu$. Hence we have

$$\begin{aligned}
A &\subseteq \alpha_\tau\left(\left\{\phi\star\mu\in\Sigma_\tau \;\middle|\; \begin{array}{l}\alpha_\tau(\phi\star\mu)\subseteq\phi^\#[\mathit{res}\mapsto S]\star\mu^\#,\\ \phi(\mathit{res})\neq\mathit{null}\end{array}\right\}\right)\\
\text{(Definition 41)} &= \alpha_\tau(\{\phi\star\mu\in\Sigma_\tau\mid\alpha_\tau(\phi\star\mu)\subseteq\phi^\#[\mathit{res}\mapsto S]\star\mu^\#\})\\
&= \alpha_\tau\gamma_\tau(\phi^\#[\mathit{res}\mapsto S]\star\mu^\#)\\
\text{(P3)} &= \phi^\#[\mathit{res}\mapsto S]\star\mu^\#\ .
\end{aligned}$$

<u>put_field</u>

Let $\tau''=\tau|_{-\mathit{res}}$ and $A=\alpha_{\tau''}(\mathsf{put\_field}_{\tau,\tau'}(\gamma_\tau(\phi_1^\#\star\mu_1^\#))(\gamma_{\tau'}(\phi_2^\#\star\mu_2^\#)))$. We have

$$\begin{aligned}
A &= \alpha_{\tau''}(\mathsf{put\_field}_{\tau,\tau'}(\{\sigma_1\in\Sigma_\tau\mid\alpha_\tau(\sigma_1)\subseteq\phi_1^\#\star\mu_1^\#\})\\
&\qquad(\{\sigma_2\in\Sigma_{\tau'}\mid\alpha_{\tau'}(\sigma_2)\subseteq\phi_2^\#\star\mu_2^\#\}))\\
&= \alpha_{\tau''}\left(\left\{\begin{array}{c}\phi_2|_{-\mathit{res}}\star\mu_2[l\mapsto\mu_2(l).\pi\star\\ \star\mu_2(l).\phi[f\mapsto\phi_2(\mathit{res})]]\end{array} \;\middle|\; \begin{array}{l}\phi_1\star\mu_1\in\Sigma_\tau,\\ \phi_2\star\mu_2\in\Sigma_{\tau'}\\ \alpha_\tau(\phi_1\star\mu_1)\subseteq\phi_1^\#\star\mu_1^\#\\ \alpha_{\tau'}(\phi_2\star\mu_2)\subseteq\phi_2^\#\star\mu_2^\#\\ (l=\phi_1(\mathit{res}))\neq\mathit{null},\\ \mu_1=_l\mu_2\end{array}\right\}\right)
\end{aligned}$$

which is $\perp$ if $\phi_1^\#(\mathit{res}) = \varnothing$, since in such a case the condition $\phi_1(\mathit{res}) \neq \mathit{null}$ cannot be satisfied. Assume then that $\phi_1^\#(\mathit{res}) \neq \varnothing$. If no creation point in $\phi_1^\#(\mathit{res})$ occurs in $\phi_2^\#|_{-\mathit{res}} \star \mu_2^\#$ then $\mu_2(l) \notin O_{\tau''}(\phi_2|_{-\mathit{res}} \star \mu_2)$. Hence the update of the content of l does not contribute to $\alpha_{\tau''}$ (Definition 41) and we have

$$A = \alpha_{\tau''}\left(\left\{ \phi_2|_{-\mathit{res}} \star \mu_2 \;\middle|\; \begin{array}{l} \phi_1 \star \mu_1 \in \Sigma_\tau,\ \phi_2 \star \mu_2 \in \Sigma_{\tau'} \\ \alpha_\tau(\phi_1 \star \mu_1) \subseteq \phi_1^\# \star \mu_1^\# \\ \alpha_{\tau'}(\phi_2 \star \mu_2) \subseteq \phi_2^\# \star \mu_2^\# \\ (l = \phi_1(\mathit{res})) \neq \mathit{null},\ \mu_1 =_l \mu_2 \end{array} \right\}\right) .$$

Let $\phi_2 \star \mu_2 \in \Sigma_{\tau'}$ be such that $\alpha_{\tau'}(\phi_2 \star \mu_2) \subseteq \phi_2^\# \star \mu_2^\#$. By P2, we can always find $\phi_1 \star \mu_1 \in \Sigma_\tau$ such that $\alpha_\tau(\phi_1 \star \mu_1) \subseteq \phi_1^\# \star \mu_1^\#$. By the hypothesis $\phi_1^\#(\mathit{res}) \neq \varnothing$ we can assume that $(l = \phi_1(\mathit{res})) \neq \mathit{null}$. If $\mu_1 =_l \mu_2$ does not hold, we can assume that $l \notin \mathrm{dom}(\mu_2)$ (up to renaming). Let $o = \mu_1(l)$. We define $\mu_2' = \mu_2[l \mapsto o.\pi \star \Im(k(o.\pi))]$. We have $\phi_2 \star \mu_2' \in \Sigma_{\tau'}$ and, since the extra location l does not contribute to $\alpha_{\tau'}$, we have $\alpha_{\tau'}(\phi_2 \star \mu_2) = \alpha_{\tau'}(\phi_2 \star \mu_2')$ and $\alpha_{\tau''}(\phi_2|_{-\mathit{res}} \star \mu_2) = \alpha_{\tau''}(\phi_2|_{-\mathit{res}} \star \mu_2')$. Moreover, $\mu_1 =_l \mu_2$ holds by construction. We conclude that the constraints on $\phi_1 \star \mu_1$ and the constraint $\mu_1 =_l \mu_2$ do not contribute to A, and we have

$$\begin{aligned} A &= \alpha_{\tau''}\left(\left\{ \phi_2|_{-\mathit{res}} \star \mu_2 \;\middle|\; \begin{array}{l} \phi_2 \star \mu_2 \in \Sigma_{\tau'}, \\ \alpha_{\tau'}(\phi_2 \star \mu_2) \subseteq \phi_2^\# \star \mu_2^\# \end{array} \right\}\right) \\ (\text{Corollary 75}) &= \xi_{\tau''}(\phi_2^\#|_{-\mathit{res}} \star \mu_2^\#) . \end{aligned}$$

Otherwise, since the objects reachable from $\phi_2(\mathit{res})$ belong to the set $O_{\tau'}(\phi_2 \star \mu_2)$, by Lemma 76 we have

$$\begin{aligned} A &\subseteq \alpha_{\tau''}\left(\left\{ \begin{array}{c} \phi_2|_{-\mathit{res}} \star \mu_2[l \mapsto \mu_2(l).\pi \star \\ \star \mu_2(l).\phi[f \mapsto \phi_2(\mathit{res})]] \end{array} \;\middle|\; \begin{array}{l} \phi_2 \star \mu_2 \in \Sigma_{\tau'}, \\ \alpha_{\tau'}(\phi_2 \star \mu_2) \subseteq \phi_2^\# \star \mu_2^\# \\ l \in \mathrm{dom}(\mu_2), \\ f \in \mathrm{dom}(F(k(\mu_2(l).\pi))) \end{array} \right\}\right) \\ &\subseteq \phi_2^\#|_{-\mathit{res}} \star \mu_2^\#[f \mapsto \mu_2^\#(f) \cup \phi_2^\#(\mathit{res})] . \end{aligned}$$

By Corollary 75 we conclude that $A \subseteq \xi_{\tau''}(\phi_2^\#|_{-\mathit{res}} \star \mu_2^\#[f \mapsto \mu_2^\#(f) \cup \phi_2^\#(\mathit{res})])$.

We prove the converse inclusion now. Since we assume that there is a $\pi \in \phi_1^\#(\mathit{res})$ which occurs in $\phi_2^\#|_{-\mathit{res}} \star \mu_2^\#$, then we can find $\phi_1 \star \mu_1 \in \Sigma_\tau$ with $\alpha_\tau(\phi_1 \star \mu_1) \subseteq \phi_1^\# \star \mu_1^\#$ and $\phi_2 \star \mu_2 \in \Sigma_{\tau'}$ with $\alpha_{\tau'}(\phi_2 \star \mu_2) \subseteq \phi_2^\# \star \mu_2^\#$ such that $\phi_1(\mathit{res}) = l$, $\mu_1(l).\pi = \pi$ and $\mu_1 =_l \mu_2$. Note that $\phi_2(\mathit{res})$ is only constrained by $\alpha_{\tau'}(\phi_2 \star \mu_2) \subseteq \phi_2^\# \star \mu_2^\#$ that is, $\phi_2(\mathit{res})$

can range over all $\phi_2^\#(res)$. Moreover, by the existence of π we can assume that l is reachable in $\phi_2 \star \mu_2$ that is $\mu_2(l) \in O_{\tau''}(\phi_2|_{-res} \star \mu_2)$. We conclude that

$$A.\mu(f) \supseteq \phi_2^\#(res). \tag{23}$$

Moreover, given again $\phi_1 \star \mu_1 \in \Sigma_\tau$ with $\alpha_\tau(\phi_1 \star \mu_1) \subseteq \phi_1^\# \star \mu_1^\#$ and $\phi_2 \star \mu_2 \in \Sigma_{\tau'}$ with $\alpha_{\tau'}(\phi_2 \star \mu_2) \subseteq \phi_2^\# \star \mu_2^\#$ and $(l = \phi_1(res)) \neq \mathit{null}$, the condition $\mu_1 =_l \mu_2$ can be made true by renaming l into l' in $\phi_2 \star \mu_2$ (if l occurs there) and extending μ_2 with an unreachable l bound to $\mu_1(l)$. We conclude that we can always find $\phi_1 \star \mu_1$ and $\phi_2 \star \mu_2$ such that $\alpha_\tau(\phi_1 \star \mu_1) \subseteq \phi_1^\# \star \mu_1^\#$, $\alpha_{\tau'}(\phi_2 \star \mu_2) \subseteq \phi_2^\# \star \mu_2^\#$, $(l = \phi_1(res)) \neq \mathit{null}$, $\mu_1 =_l \mu_2$ and l is not reachable from $\phi_2 \star \mu_2$: $\mu_2(l) \notin O_{\tau''}(\phi_2|_{-res} \star \mu_2)$. As a consequence and by using P2, we have

$$\begin{aligned} A &\supseteq \alpha_{\tau''}\left(\left\{ \phi_2|_{-res} \star \mu_2 \,\middle|\, \begin{array}{l} \phi_2 \star \mu_2 \in \Sigma_{\tau'}, \\ \alpha_{\tau'}(\phi_2 \star \mu_2) \subseteq \phi_2^\# \star \mu_2^\# \end{array} \right\}\right) \\ \text{(Corollary 75)} &= \xi_{\tau''}(\phi_2^\#|_{-res} \star \mu_2^\#) \ . \end{aligned} \tag{24}$$

By merging (23) and (24) we conclude that

$$A \supseteq \xi_{\tau''}(\phi_2^\#|_{-res} \star \mu_2^\#) \cup (\phi_\perp \star \mu_\perp[f \mapsto \phi_2^\#(res)]) \tag{25}$$

where $\phi_\perp$ maps all variables to $\varnothing$ and $\mu_\perp$ maps all fields to $\varnothing$. We still have to prove that in the equation above we can move $\phi_2^\#(res)$ inside the garbage collector $\xi_{\tau''}$. But this is true since by Figure 7 we know that f is a field of $F(\tau(res))$ so that f is a field of the objects created at the creation point $\pi \in \phi_1^\#(res)$ which we assume to occur in $\phi_2^\#|_{-res} \star \mu_2^\#$. Hence $\xi_{\tau''}$ cannot garbage collect the set $\phi_2^\#(res)$ bound to f. In conclusion, (25) becomes

$$A \supseteq \xi_{\tau''}(\phi_2^\#|_{-res} \star \mu_2^\#[f \mapsto \mu_2^\#(f) \cup \phi_2^\#(res)]).$$

$\underline{\cup}$
By additivity (Proposition 49), the best approximation of $\cup$ over $\wp(\Sigma_\tau)$ is (pointwise) $\cup$ over $\mathcal{ER}$.

The proof of Proposition 56 needs the following result that $\theta_\tau(e)$ is an element of $\mathcal{ER}_\tau$ and approximates exactly the same concrete states as e.

LEMMA 78 *Let $\sigma \in \Sigma_\tau$ and $e \in \mathcal{E}_\tau$. Then $\theta_\tau(e) \in \mathcal{ER}_\tau$. Moreover, $\alpha_\tau^{\mathcal{E}}(\sigma) \subseteq e$ if and only if $\alpha_\tau^{\mathcal{ER}}(\sigma) \subseteq \theta_\tau(e)$.*

Proof. We have $\theta_\tau(e) \in \mathcal{ER}_\tau$ by idempotency of ξ_τ (Proposition 46) and Definition 48.

Let $\alpha^{\mathcal{E}}_\tau(\sigma) \subseteq e$ and $v \in \mathsf{dom}(\tau)$. If $\tau(v) = \mathit{int}$, then $\varepsilon_\tau(\sigma)(v) = * = \vartheta_\tau(e)(v)$. If $\tau(v) \in \mathcal{K}$, then every $\pi \in \varepsilon_\tau(\sigma)(v)$ is such that $k(\pi) \leq \tau(v)$ (Definitions 37 and 14). Moreover, $\pi = \mu(l).\pi$ for some $l \in \mathsf{rng}(\phi) \cap \mathit{Loc}$ (Definition 37). Hence $\pi \in \alpha^{\mathcal{E}}_\tau(\sigma)$ (Definition 22), and $\pi \in e$. By Definition 54 we conclude that $\pi \in \vartheta_\tau(e)(v)$. Hence $\alpha^{\mathcal{ER}}_\tau(\sigma).\phi = \varepsilon_\tau(\sigma) \subseteq \vartheta_\tau(e)$. Let now $f \in \mathsf{dom}(\widetilde{\tau})$. If $\widetilde{\tau}(f) = \mathit{int}$, then $\varepsilon_{\widetilde{\tau}}(\{\widetilde{o.\phi \star \mu} \mid o \in O_\tau(\sigma)\})(f) = * = \vartheta_{\widetilde{\tau}}(e)(f)$. If $\widetilde{\tau}(f) \in \mathcal{K}$, then every $\pi \in \varepsilon_{\widetilde{\tau}}(\{\widetilde{o.\phi \star \mu} \mid o \in O_\tau(\sigma)\})(f)$ is such that $k(\pi) \leq \widetilde{\tau}(f)$ (Definitions 37 and 14). Moreover, $\pi = \mu(l).\pi$ for some $l \in \mathsf{rng}(o.\phi) \cap \mathit{Loc}$ with $o \in O_\tau(\sigma)$ (Definition 37). Hence $\pi \in \alpha^{\mathcal{E}}_\tau(\sigma)$ (Definition 22), and $\pi \in e$. By Definition 54 we conclude that $\pi \in \vartheta_{\widetilde{\tau}}(e)(f)$. Hence $\alpha^{\mathcal{ER}}_{\widetilde{\tau}}(\sigma).\mu = \varepsilon_{\widetilde{\tau}}(\{\widetilde{o.\phi \star \mu} \mid o \in O_\tau(\sigma)\}) \subseteq \vartheta_{\widetilde{\tau}}(e)$. In conclusion, we have $\alpha^{\mathcal{ER}}_\tau(\sigma) \subseteq \vartheta_\tau(e) \star \vartheta_{\widetilde{\tau}}(e)$. Since ξ_τ is monotonic (Proposition 46) and by Proposition 47, we have $\alpha^{\mathcal{ER}}_\tau(\sigma) \subseteq \xi_\tau(\vartheta_\tau(e) \star \vartheta_{\widetilde{\tau}}(e)) = \theta_\tau(e)$.

Conversely, let $\alpha^{\mathcal{ER}}_\tau(\sigma) \subseteq \theta_\tau(e)$. Let $\pi \in \alpha^{\mathcal{E}}_\tau(\sigma)$. By Definition 22 we have $\pi = o.\pi$ with $o \in O_\tau(\sigma)$. By Definition 41 we have $\pi \in \alpha^{\mathcal{ER}}_\tau(\sigma).\phi(v)$ for some $v \in \mathsf{dom}(\tau)$ or $\pi \in \alpha^{\mathcal{ER}}_\tau(\sigma).\mu(f)$ for some $f \in \mathsf{dom}(\widetilde{\tau})$, and hence $\pi \in \theta_\tau(e).\phi(v)$, in the first case, or $\pi \in \theta_\tau(e).\mu(f)$, in the second case. In both cases, by Definition 54 we have $\pi \in e$. Thus $\alpha^{\mathcal{E}}_\tau(\sigma) \subseteq e$. □

We can now prove that every element of $\mathcal{E}$ represents the same set of concrete states as an element of $\mathcal{ER}$.

Proposition 56. *Let $\gamma^{\mathcal{E}}_\tau$ and $\gamma^{\mathcal{ER}}_\tau$ be the concretisation maps induced by the abstraction maps of Definitions 22 and 41, respectively. Then $\gamma^{\mathcal{E}}_\tau(\mathcal{E}_\tau) \subseteq \gamma^{\mathcal{ER}}_\tau(\mathcal{ER}_\tau)$.*

Proof. By Lemma 78, for any $e \in \mathcal{E}_\tau$, we have

$$\begin{aligned}\gamma^{\mathcal{E}}_\tau(e) &= \{\sigma \in \Sigma_\tau \mid \alpha^{\mathcal{E}}_\tau(\sigma) \subseteq e\} \\ &= \{\sigma \in \Sigma_\tau \mid \alpha^{\mathcal{ER}}_\tau(\sigma) \subseteq \theta_\tau(e)\} = \gamma^{\mathcal{ER}}_\tau(\theta_\tau(e)).\end{aligned}$$

Since this holds for all $e \in \mathcal{E}_\tau$, we have the thesis. □